National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

ACTION DIAGRAMS: A METHODOLOGY FOR THE SPECIFICATION AND VERIFICATION OF REAL-TIME SYSTEMS

by

Karim Khordoc

Department of Electrical Engineering
McGill University, Montreal
March, 1996

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Doctor of Philosophy

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

ISBN 0-612-12400-2

Canada

To my wife Marie-Claude
and to my children Patrick, Philip, and Valerie

ABSTRACT

In this thesis, we address issues in the specification, simulation, and formal verification of systems that are characterized by real-time constraints and a mix of protocol and data computation aspects. We propose a novel specification language and modeling methodology - HAAD (Hierarchical Annotated Action Diagrams). In HAAD, the interface behavior of a system is captured as a hierarchy of action diagrams. The internal behavior is modeled by an Extended Finite State Machine (EFSM). A leaf action diagram defines a behavior (a template) over a set of ports. Procedures and predicates are attached to actions in order to describe the functional aspects of the interface.

We propose algorithms and methods for the automatic generation of simulation models and response verification scripts from HAAD specifications. These models perform "on-the-fly parsing" of actions received at their I/O ports, sequencing through state transitions based on the result of this parsing, detecting incorrect, or ill-formed interface operations (bus cycles), verifying that all timing constraints at the input of the model are met, and driving the model outputs with appropriate delays.

We formalize the operational semantics of leaf action diagrams under linear timing constraints, based on the concepts of a *block machine* and *causal* block machine. We state the realizability of an action diagram in terms of the existence of a causal block machine derived from the action diagram. We examine the problem of the compatibility of concurrent, communicating leaf action diagrams described by linear timing constraints and we show the inaccuracies of known methods that address this problem. We define the action diagram compatibility problem in terms of the compatibility of *all* the possible combinations of causal block machines derived from these action diagrams. We prove that such enumeration is not needed in answering the compatibility question. This leads to an exact and efficient compatibility verification procedure.

RÉSUMÉ

Dans cette thèse, nous traitons de la problématique de la spécification, simulation, et vérification formelle de systèmes caractérisés par des contraintes en temps réel et par un mélange d'aspects de protocoles et de traitement de données. Nous proposons un nouveau langage de spécification et une méthodologie de modélisation - HAAD (Hierarchical Annotated Action Diagrams - Diagrammes d'Actions Annotés Hiérarchiques). En HAAD, le comportement à l'interface d'un système est représenté par une hiérarchie de diagrammes d'actions. Le comportement interne du système est représenté par une machine à états finis étendue. Un diagramme d'actions feuille définit un comportement (un gabarit) sur un ensemble de ports. Des procédures et des prédicats sont attachés aux actions afin de décrire l'aspect fonctionnel de l'interface.

Nous proposons des algorithmes et des méthodes pour la génération automatique, à partir de spécifications HAAD, de modèles de simulation et de scripts de vérification des réponses du système. Ces modèles traitent "à la volée" les actions reçues sur leurs ports d'entrées / sorties, accomplissent le séquencement d'états approprié, détectent les opérations d'interface (cycles de bus) mal formées, vérifient que toutes les contraintes temporelles aux entrées du modèle sont respectées, et contrôlent les sorties du modèle moyennant les délais appropriés.

Nous procédons à la formalisation de la sémantique opérationnelle des diagrammes d'actions feuille sous contraintes temporelles linéaires. Cette formalisation est basée sur les concepts de *machine à blocs* et machine à blocs *causale*. Nous formulons la réalisabilité d'un diagramme d'actions en terme de l'existence d'une machine à blocs causale dérivée du diagramme d'actions. Nous examinons le problème de la compatibilité de diagrammes d'actions communicants décrits par des contraintes temporelles linéaires, et nous montrons l'inexactitude des méthodes connues traitant ce problème. Nous définissons le problème de compatibilité de diagrammes d'actions en terme de la compatibilité de *toutes* les combinaisons possibles de machines à blocs causales dérivées de ces diagrammes d'actions. Nous faisons la preuve que cette énumération n'est pas nécessaire pour répondre à la question de compatibilité. Ceci donne lieu à une procédure exacte et efficace de vérification de la compatibilité.

ACKNOWLEDGMENTS

I am deeply grateful to my thesis supervisors, Drs Nicholas Rumin and Eduard Cerny for providing me with the opportunity to go through the Ph.D. program. Working closely with Dr Eduard Cerny has been a fulfilling and rewarding experience. I cannot thank Dr Cerny enough for his relentless energy, motivation and patience in supervising this work.

Grateful acknowledgments are also made to:

Engineering managers at Bell-Northern Research Ltd., Mr Allan Silburt, Mr Robert Hum and Mr Philip Pownall for their helpful discussions, encouragements, and continuous support of this research.

Former M.Sc. students at the Université de Montréal, Mr Mario Dufresne, Mr Philippe-André Babkine, Mrs Simona Gandrabur, and Mr Andrei Tarnauceanu, for their efforts in carrying the detailed design and software implementation of the HAAD specification and simulation packages.

Former post-doctoral fellow Dr Tahar AliYahia for his assistance in the design and software implementation of the formal static timing verification package for leaf action diagrams.

The Natural Sciences and Engineering Research Council of Canada (NSERC) and Bell-Northern Research Ltd. (BNR) for their financial support of this research.

I would also like to thank my wife Marie-Claude for sharing the dream and for her moral support and love. Last, but not least, I am eternally grateful to my parents who have given me a solid foundation of love, trust and the desire to pursue success and happiness.

REMARKS CONCERNING THESIS PREPARATION

In accordance with the *Guidelines for Thesis Preparation* (September 1994 revision) of the Faculty of Graduate Studies and Research, McGill University, the following text is cited:

"Candidates have the option of including, as part of the thesis, the text of a paper(s) submitted or to be submitted for publication, or the clearly-duplicated text of a published paper(s). These texts must be bound as an integral part of the thesis.

If this option is chosen, connecting texts that provide logical bridges between the different papers are mandatory. The thesis must be written in such a way that it is more than a mere collection of manuscripts; in other words, results of a series of papers must be integrated.

The thesis must still conform to all other requirements of the *Guidelines for Thesis Preparation*. The thesis must include: A Table of Contents, an abstract in English and French, and introduction which clearly states the rationale and objectives of the study, a comprehensive review of the literature, a final conclusion and summary, and a thorough bibliography or reference list.

Additional material must be provided where appropriate (e.g., in appendices) and in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported in this thesis.

In the case of manuscripts co-authored by the candidate and others, the candidate is required to make an explicit statement in the thesis as to who contributed to such work and to what extent. Supervisors must attest to the accuracy of such statements at the doctoral oral defense. Since the task of the examiners is made more difficult in these cases, it is in the candidate's interest to make perfectly clear the responsibilities of all the authors of the co-authored papers. Under no circumstances can a co-author of any component of such a thesis serve as an examiner for that thesis."

This thesis consists of seven chapters and three appendices. Chap-

ters 2 to 6 are in the form of papers, published or submitted for publication. Appendix III states for each paper, where and when it was published or submitted, and what the co-author contributions were. Chapter 1 contains the connecting texts that provide logical bridges between the different papers.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

CHAPTER 2

A STIMULUS / RESPONSE SYSTEM BASED ON HIERARCHICAL TIMING DIAGRAMS

CHAPTER 3

MODELING AND EXECUTION OF TIMING DIAGRAMS WITH OPTIONAL AND MULTI-MATCH EVENTS

CHAPTER 4

INTEGRATING BEHAVIOR AND TIMING IN EXECUTABLE SPECIFICATIONS

CHAPTER 5

MODELING CELL PROCESSING HARDWARE WITH ACTION DIAGRAMS

CHAPTER 6

SEMANTICS AND VERIFICATION OF ACTION DIAGRAMS WITH LINEAR TIMING CONSTRAINTS

CHAPTER 7
GENERAL CONCLUSIONS

APPENDIX I

SYNTACTIC WELL-FORMEDNESS RULES FOR ACTION DIAGRAMS

APPENDIX II

THE DEFBEHAVIOR LANGUAGE

APPENDIX III
CO-AUTHORS' STATEMENT

CHAPTER 1

INTRODUCTION

1 Problem Description

Due to the increasing complexity of digital systems and to competitive market pressures, the digital systems industry has witnessed a marked shift towards higher abstraction levels in the areas of modeling, verification, and synthesis. Higher-level modeling allows to remove ambiguity from system specifications. It also allows the designer to concentrate on the "bigger picture", rather than getting distracted by details. Furthermore, it is the starting point for verification and synthesis from higher levels. Verification at higher levels allows to find design problems earlier. It also enables the verification of much more complex systems than would have been possible otherwise. Synthesis from high levels is the key to increased automation, and hence to productivity gains in the design process.

In this thesis, we address issues in high-level modeling and verification. We concentrate on systems that have real-time requirements and that present a mix of protocol aspects and data computation aspects. The problems addressed by the thesis are summarized in the following.

Problem: modeling and analysis of real-time systems

Systems that have real-time requirements and a mix of protocol aspects and data processing aspects are difficult to design correctly and verify. When these aspects are intermixed in a computer model of the system, the model typically becomes hard to understand and too complex to analyze by computer-aided design (CAD) tools. A more practical approach is to use dedicated CAD tools and techniques to separately verify different aspects (e.g., data processing functions versus protocol handlers) and

different levels of abstractions of the system behavior. This, however, is possible only if the modeling methodology allows such separation. There is currently a need for such methodologies.

Problem: system integration

It has often been reported in industry [1] that a large proportion of the failures that are found in an ASIC (Application Specific Integrated Circuit) after its fabrication are in fact discovered *after* the ASIC has been integrated in the system that it is intended to work with. In addition, many of these failures are caused by ambiguous specifications of the interface protocols that govern the transactions between the ASIC and the rest of the system. According to professionals in the EDA (Electronic Design Automation) industry [2, 3], there currently is a pressing need for tools and methodologies that could help alleviate these problems.

Problem: test bench development time

In a typical state of the art ASIC based system design environment, designers spend anywhere from 25% [1] to 65% [4] of their time developing "test benches". These are software procedures that run concurrently with the (sub-)system model in a simulation environment. The test bench stimulates the model and verifies its responses against the specifications. Due to some of its aspects that are related to the engineering of communication protocols, test bench development is an error-prone task. The software code involves process synchronization primitives (e.g., WAIT statements) and is hard to debug. It is also difficult to ensure that the test bench is complete, i.e., whether all the properties that need to be concurrently verified in a given execution scenario, are indeed checked for. Due to a lack of structured approach to test bench development (often compounded with the inherently ad-hoc nature of the set of properties to be verified), the resulting software is hard to maintain.

For the above reasons, and due to the fact that test bench software represents as much as 50% of the total software written for a hardware development project [1], test bench development in its present form puts a substantial burden throughout the life cycle of the product design data.

Problem: linear timing constraints in interface specifications

When designing a component that is intended to operate in a distributed real-time system, the designer must make sure that the interacting components of the system have compatible interface protocols, i.e.,

that each system component satisfies the rules and assumptions that the other components make on their environment. The most commonly used compatibility verification techniques are visual inspection and simulation. However, due to the often high degree of concurrency in a distributed system and due to the min-max intervals that characterize the delays and timing assumptions in the specifications, the number of cases that needs to be considered can be overwhelming for computer simulation (let alone visual inspection). Hence, the interest in potentially more reliable techniques, i.e., formal verification [5, 6, 7, 8, 9].

The timing specifications of interface protocols are often described by linear constraints. These capture in a declarative and abstract way the set of allowed behaviors and assumptions of the component. This description style decouples the specification from the implementation, thus leaving more flexibility to the interface designer. This decoupling is also desirable to vendors publishing the interface specifications of their proprietary products.

The problem is, however, that linear timing constraints can make an interface specification *non-causal*, in the sense that the interface can be implemented only by a system that "guesses" the future behavior of other components that interact with it. Non-causality can manifest itself even when the constraint system is consistent (i.e., its solution set is non-empty). In addition, non-causality can invalidate the outcome of known compatibility verification procedures [6]. To the best of our knowledge, there does not exist a tool or methodology that correctly answers the interface compatibility question in the presence of linear timing constraints.

2 Relevant Work

2.1 Modeling

Behavioral modeling approaches, such as [10, 11], lack the timing constraint constructs and the capability of declaring the assumptions that a behavior makes on its environment.

Timing diagrams [12] and message sequence charts [13] are event [1] (action) based notations that are widely used in the hardware design community, as well as the communication protocols and distributed systems design communities. These notations are of a declarative nature. They are

[1] In this thesis, the terms *event* and *action* are used interchangeably.

convenient for describing families of execution scenarios in terms of event sequences over time. The notations emphasize the abstract specifications view of a system, rather than its implementation details In [14], the timing diagram notation is formalized, and its expressive power extended. Event values and state variables can be expressed using "extended boolean expressions" on signals; in addition to the standard boolean connectives, these include *Delay* and *Latch* constructs. Looping and conditional executions of timing diagrams are supported using extended boolean expressions to control the execution. Timing diagrams can be combined concurrently by specifying synchronization constraints between events in different diagrams. The captured specifications are used for the synthesis of interface circuits.

Interface specifications describe the protocols that govern the interactions between the components of a system. For example, interactions over a hardware bus consist of operational units called "interface operations", or "bus cycles", such as FETCH, READ, WRITE cycles, etc. Each interface operation consists of specific event sequences related by timing constraints. At a higher level of abstraction, e.g., in modeling a distributed computer system, the operational units are system transactions, e.g., file transfer operations in which the events model remote procedure calls, connection/ disconnection requests and acknowledgments, start/end of data transfers etc.

In its simplest form, an interface specification is represented by a *timing constraint graph* [15]. This is a weighted directed graph in which vertices represent interface events and a directed edge of weight Δ_{ij} from a vertex a_i to a vertex a_j represents the linear timing constraint $t(a_j)-t(a_i) \leq \Delta_{ij}$, where $t(a_i)$ and $t(a_j)$ are the occurrence times of events a_i and a_j, respectively, and Δ_{ij} is a constant. In [16], the model is extended to *latest* and *earliest* constraints. An event related to its causal predecessors by latest (earliest) constraints will occur only after (as soon as) the last (first) of the predecessors have occurred. In [5], the behavior of an interface is expressed as a set of event occurrence rules. Each such rule is described by a cause-effect relationship and a delay interval between two events. Optionally, a boolean expression on signal states specifies the condition under which a rule applies. Note, however, that the timing relationships that can be expressed in this framework are too simple to exhibit the causality [2] problem mentioned in the Section 1.

Interface specifications must be related to the internal aspects of

[2]The term "causality" in [5] simply indicates the cause-effect nature of the event occurrence rules.

behavior and to structure. These relations are traditionally of concern to behavioral synthesis systems. For example, in [17], both the interface specifications (captured by timing diagrams) and data-flow specifications (captured by a textual HDL description) are described in a unified graph in which nodes represent data-flow operations and interface events, and arcs represent data dependencies and timing constraints. Data dependency arcs between input/output event nodes and operation nodes capture the interrelation between interface and internal behavior. From a specification point of view, the interface and data-flow descriptions are related only through I/O signal names and symbolic data names (i.e., common name space between the two specifications for I/O signals and symbolic values on data busses). As a result, the HDL specification contains control-flow information which could be redundant with respect to that captured in the interface specification. [18] extends the work of [17] by including structural domain descriptions in the unified graph: event nodes can be grouped into "wires" and operation nodes can have either wires or events as their input/outputs. Also, a more powerful description of event dependencies and timing constraints is supported using a subset of first-order predicate calculus. The *Design Data Structure* (DDS) representation of [19] consists of three separate graphs: Data-Flow Graph (DFG), Control and Timing Graph (CTG), and Structure Graph (SG). The graphs are related by "bindings", e.g., the scheduler of the synthesis system binds an operation of the DFG to an interval arc of the CTG. Causal relations and timing constraints can be specified between interface events in the CTG. An interface event can be bound to an interval arc and to a destination node in the CTG; the arc specifies the time interval in which the event can occur and the node indicates the destination control point to which processing will branch if the event occurs. In addition, a boolean expression can be associated with the event to specify the condition under which the event can occur.

More complex interfaces as well as control-oriented real-time systems and protocol handlers can be described as timed, communicating or concurrent abstract entities, each consisting of timed event sequences, state-dependent causality relations between events, and assertions on state changes due to event occurrences, and timing requirements. For example, in [20], the author argues for a specification methodology in which a high-level implementation of a system is described as a set of communicating processes described at the extended state machine level and the properties (or requirements) that the system must satisfy are described in a declarative style as a set of *event expressions* in a special-purpose timed logic designated as *CPA* (Conditionals, Precedence relations, Assertions). Each

event expression in CPA consists of a *precedence* relation defining an ordering between two or more events, a logic *condition* under which the expression applies, and a logic *assertion* specifying constraints on the sequence numbers, values and times of the events named in the expression. Both the condition and assertion are expressed in first-order predicate calculus over events, their values and their times. Minimum and maximum timing constraints can be specified. Events can be identified by indices referring to particular instances of their occurrence (e.g., in the case of repetitive events); the indices can be absolute or relative to a designated reference event in the event expression. Hierarchy is introduced by specifying *super-events* which are sequences of atomic events.

In [21], the properties that a system must satisfy are expressed in a subset of real-time temporal logic (RTTL) [22] (this subset is limited to properties describing invariance and/or real-time response). The system itself is described by a finite-state Timed Transition Model (TTM). A TTM is characterized by a set of variables and a set of transitions that modify these variables. Each transition is characterized by an enabling pre-condition (i.e., a boolean expression on the TTM variables), lower and upper time bounds for the delay from the enabling of the transition (when the pre-condition becomes true) to its actual firing, and a set of post-actions (modifications of the TTM variables) that take place upon firing of the transition. The firing semantics are similar to those used in time Petri nets [23].

2.2 Simulation

Simulation techniques [24, 25, 26] are very useful in exercising the system specifications. *Interface simulation models* are behavioral HDL programs derived from the interface specifications of the components that form the system's environment. The interface simulation model of a component consists in "on-the-fly parsing" of events received at the component's I/O ports, sequencing the model through its state transitions based on the result of this parsing, detecting incorrect, or ill-formed interface operations (bus cycles), verifying that all timing constraints at the input of the component are met, and driving the component outputs with appropriate delays.

The *HIDE* system [27] generates VHDL interface models from timing diagrams and state diagrams. The state diagrams specify the interface control-flow. A VHDL procedure is generated for each interface operation (such as READ, WRITE etc.). The procedures can then be called from a

command file to simulate the interface behavior. This approach, however, does not seem to be practical for cases such as memory devices, wherein the choice of the actual interface operation cannot be decided before-hand (i.e., the interface control-flow is governed by the environment, e.g., the processor).

In [28], a VHDL annotation language, *VAL+*, is proposed to describe parameterized, hierarchical event patterns. The patterns are used for matching simulation traces; the idea is to transform (flat) simulation traces into hierarchical ones, by pattern matching, in order to help the user in trace debugging and browsing. However, the matching is done off-line, after the simulation has completed; this requires the storage of the complete simulation trace. In addition, the patterns are used only for trace matching, not for driving the circuit under simulation.

2.3 Formal Verification

The advantage of the simulation techniques outlined in the previous paragraph is that they handle large and complex models. However they only provide a partial "coverage" with respect to the model being verified. Complementary techniques that are starting to emerge in the digital design industry are based on formal methods. These techniques can be seen as "orthogonal" to the techniques of the previous paragraph in that they can provide *complete* coverage of a *partial* model. In this section, we review some of the formal techniques that are relevant to real-time systems and interface verification.

One way to decompose the interface verification problem is to examine "interface scenarios", i.e., finite unrolled behaviors [15, 5, 6, 7, 8, 9].

A finite interface scenario described by linear timing constraints is consistent if there are no cycles of negative weight in the corresponding constraint graph [15, 6]. In [15], a constraint priority scheme defined by the user, is used to relax some constraints, thus removing inconsistencies from the interface specification. In [5] where logic conditions can qualify constraints, the system checks for the logic consistency of paths. However concurrent state changes of side path variables (i.e., signals that have no associated events on the considered causal path) are not taken into account, thus possibly resulting in erroneous analysis.

In [6], the authors propose a method based on the shortest path algorithm [29] for the verification of the interface compatibility of two communicating system components described by timing diagrams under

linear timing constraints. However, their method is too pessimistic (i.e., it can yield false negative answers to the compatibility question), unless the communication between the system components is unidirectional (i.e., one component has no input events, and the other has no output events). Other works address the issue of efficient algorithms for computing the maximal time distances between events for more complex forms of timing constraints in timing diagrams [7, 8]. For example, efficient methods exist for computing the shortest distances over linear and max *latest* constraint systems [7, 9]. The inclusion of *earliest* constraints makes the problem of computing time distances between events NP-complete [7]. In [9], the authors show how a Constraint Logic Programming (CLP) environment based on relational interval arithmetics (RIA) [30] can be used to solve the maximal time distance problem in the cases of 1- linear constraints only, 2- max-only or min-only constraints, and 3- linear constraints intermixed with either max or min constraints. They show that for these three cases the general CLP/RIA approach has the same worst case time complexity as the ad-hoc approach of [7]. An additional advantage of the CLP/RIA approach is that, due to its general purpose nature, it is a better vehicle for extensions to the basic problem, e.g., accounting for delay correlations, or annotating constraints with logic (boolean) conditions in a unified computational framework. In [31], the authors solve the maximal time distance computation in cyclic (process like) timing diagrams with max only constraints (also designated as constraints of the *latest* type, i.e., an event occurs only after the last of its predecessors has occurred). A similar problem is solved by Escalente et al. [32] using a combination of graph-based and linear programming techniques. The authors state that their approach can be generalized to the mixed min/max problem, but they do not sufficiently elaborate on that.

None of the methods mentioned in the previous paragraph address the issue of *realizability* of timing diagram specifications, i.e., can the specification be simulated by a causal system. Due to their declarative style (as opposed to e.g., an operational style), linear constraints make the causality issue a non-trivial one. In practice, synthesis methods such as [14] that do not examine the causality issue under linear constraints, may produce systems that only satisfy mutually incompatible subspaces of their respective specifications. The consequence is the risk of incompatibility between independently developed implementations of the interacting systems. In [33], the authors define a realizability criterion called well-posedness. However, it turns out that this criterion is not sufficiently powerful for reasoning on some of the practical examples that we examined (e.g., interface operations of a Motorola MC68360 processor). Recently, timed process algebras have

emerged [34] in which the occurrence times of events can be related by linear conjunctive constraints. However, the underlying semantic models proposed in these works do not address the causality issue. Hence, such methods do not reveal whether the specified system can be built from independently developed subsystems, each constructed according to its local specification.

In [21], an automatic procedure is given for verifying whether a system described as a Timed Transition Model satisfies a formula in a subset of Real-Time Temporal Logic (RTTL). A reachability graph is constructed, on which the RTTL formula is then checked for validity. In this approach, the number of states in the reachability graph grows very rapidly, due to two factors. First, states are created in the reachability graph for *every* time point in the analysis (time is considered as a TTM variable which is incremented by unit "ticks"). Second, *all* real-time realizable total orders of transition firings are enumerated, whether or not they affect the validity of the formula under verification. Other verification approaches based on timed extensions of process algebras [35, 36] or on time Petri nets [23] are characterized by similar complete state enumeration.

In [37], the system and the properties to verify are described as an interconnection of units forming a closed system. Each unit is a "Time Sequential Machine". Similarly to [21], a delay bound is associated with each state transition of a unit. Failure to satisfy a property is indicated by a given "checking" unit going into an *Error* state. A partial order approach is used in the reachability analysis of the closed system, thus avoiding the enumeration of all the possible interleavings of the state transitions in the units. Furthermore, timing relationships between transitions are compactly represented by a timing constraint graph, therefore avoiding the creation in the reachability graph of state nodes for each time "tick".

A timed automaton is, strictly speaking, an infinite state system (due to the continuous time model assumption). In [38], the timed automaton concept is formalized and the author proves that, even under a continuous time model, there exists a *finite* representation of the state space of the original timed automaton. This representation is based on a concept known as *region automaton*, wherein each "region" is an equivalence class with respect to the property being verified. Each region is associated with a finite constraint graph that implicitly represents the (generally infinite) number of states in the given equivalence class. In [39], a branching real-time temporal logic, TCTL, is defined, and the traditional model-checking procedure [40] is extended to allow checking of the validity of a TCTL formula on a timed automaton.

In [41], the semantics of a subset of our timing diagram model are defined in terms of a timed process algebra, TDA, based on the works of [34] and [42]. Then, a procedure is given for translating a term of the algebra into a timed automaton. The timed automata resulting from individual terms are hierarchically composed to obtain the final timed automaton.

Another technique that efficiently exploits constraint graphs in the formal analysis of real-time behavior is given in [43]. The system verifies whether a specification satisfies a given safety assertion, where both the specification and the safety property are described in a subset of Real Time Logic (RTL) [44]. However, there is no explicit concept of state in the specification paradigm and therefore the method is inadequate for describing systems with state dependent behavior.

Simulation and formal verification can be advantageously combined in a unified environment for the analysis of communication protocols. For example, in [45] a complete verification with respect to a set of properties is done on a simplified model of the interacting protocols and simulation is performed on a detailed model. Simulation traces are analyzed on-the-fly by an "observer" program that is automatically compiled from a declarative specification of the properties to be verified, given in first-order predicate logic.

3 Original Contributions

In this section, we summarize our original contributions towards solving the problems of Section 1, and we put these contributions in the context of the other works discussed in Section 2. The original contributions of the thesis fall into three categories: 1- modeling language and methodology for real-time systems, 2- executable model generation, and 3- formal timing verification.

1- In the area of modeling language and methodology for real-time systems, the original contributions are:

- **Separation of, and links between, interface behavior and internal behavior:**

  We propose a novel interface modeling methodology, HAAD - Hierarchical Annotated Action Diagrams in which the interface behavior is captured separately from the internal behavior while maintaining the links between the two. The interface behavior is captured as a

hierarchy [3] of action diagrams. We assume that the internal behavior is modeled by an Extended Finite State Machine (EFSM). We propose to link the interface behavior and internal behavior by shared variables and *synchronization points*. This modeling methodology facilitates the verification of the interface behavior and should also facilitate that of the internal[4] behavior.

- **Separation of, and links between, functional aspects and protocol/ timing aspects in interface specifications:**

  One of the main novelties of HAAD is that the data manipulation aspects of an interface specification are "overlaid" onto the hierarchy of action diagrams. This overlay is in the form of HDL procedures, functions and variables that are attached to actions (designated as *trigger actions*) of the action diagram hierarchy. The procedures and functions are executed when their trigger actions occur. State variables that are attached to actions provide data-flow links between the data manipulation procedures and the action diagram protocol description. This approach facilitates the verification of the system. In contrast, when modeling interfaces in plain HDL, the timing and protocol behavior is intermingled with the functional behavior. In addition, there is no possibility of expressing protocol rules and timing constraints, except by writing procedural checkers for them (and in that case, the "how" of rule checking would be captured instead of the "what" of the rules themselves). Compared to [17], our approach is based on directly linking data-flow operations to interface actions. This avoids the description redundancies of [17].

- **Combination of a true behavioral hierarchy *and* a rich set of timing constructs:**

  HAAD is the first modeling language that combines a true behavioral hierarchy and a rich set of timing constructs. In this hierarchy, behaviors are composed using operators such as *Concatenation*, *Choice*, *Concurrency*, *Loop* and *Exception-Handler*. Port maps and parameter maps specify how the operators combine the behaviors. Actions in leaf action diagrams can be related by weighted (min/max) *timing constraints*. The constraints are of *assume* or *commit* intent, and

[3] Here we are using the term *action diagram* to generically indicate any component (whether leaf or not) in this hierarchy. The leaves of a HAAD hierarchy resemble the more familiar timing diagrams. For historical reasons, in the body of the thesis, the terms *action diagram* and *timing diagram* are used interchangeably.

[4] Note that internal behavior verification is not explored in this thesis.

they can be combined to form more complex constraints using the *conjunctive*, *earliest* and *latest* composition operators. Other works that put the emphasis on behavioral hierarchy ignore the timing aspects, e.g., [11], or offer only rudimentary timing support, e.g., [10]. On the other hand, works that concentrate on timing specifications ignore behavioral composition [14].

- **Delayed choice semantics:**

  HAAD is the first modeling language that proposes the concept of a *delayed choice*, whereby the selection of a behavior (choice branch) is delayed until sufficient information is gathered. This is useful in supporting the concept of interface operations in "scenario-based" modeling.

2- In the area of executable model generation, the original contributions are:

- **Dynamic stimulus generation and response validation from timing diagrams:**

  Our work [46] is the first to report on the automatic generation of simulation models and response verification scripts from action diagrams. The advantage of this capability is to markedly accelerate the test bench development process. In addition, since the designer is now relieved from many of the low-level details of test bench development, he/she can concentrate more effectively on what needs to be verified, rather than how to verify it.

- **Unified framework for *valid* and *don't care* signal states:**

  In order to handle *valid* and *don't care* signal states in a unified modeling and execution framework, we introduce two new action types: *optional* and *multi-match* actions [47]. Simpler alternative approaches, e.g., "data" *stability windows* with respect to "clock" and "control" signals, are not general enough for expressing complex timing specifications, e.g., asynchronous RAMs [48]. A concept similar to a multi-match action was proposed in [49] for the synthesis of asynchronous circuits from Signal Transition Graphs, however our work is the first to consider optional and multi-match actions in the generation of simulation models and response verification scripts.

- **Unified approach to master, slave and mixed behaviors:**

  Another novel aspect in our test bench and model generation approach is that it is independent of whether the modeled system is a master (i.e., autonomously generates requests), slave (i.e., services requests), or mixed (i.e., exhibits a combination of both master and slave characteristics). In other works, e.g., [27], HDL procedures are generated for each interface operation (such as READ, WRITE etc.). The procedures can then be called from a command file to simulate the interface behavior. This approach, however, is not suited to behaviors in which the choice of the actual interface operation cannot be decided before-hand (e.g., a slave type of behavior).

3- In the area of formal timing verification, the original contributions are:

- **Sufficient conditions for the well-behavedness of interface specifications under linear timing constraints:**

  Our work is the first to propose technology independent sufficient conditions for the well-behavedness of interface specifications under linear timing constraints, such that these conditions: 1- guarantee that the specifications can be simulated by a causal system, and 2- are general enough to handle the complex timing of bus interface specifications. We show that the interface *consistency* criterion used in other interface verification works, e.g., [6], or in interface synthesis e.g., [14], is not a sufficient well-behavedness criterion, while the *well-posedness* criterion of [33] is not general enough for some commonly used bus interfaces.

- **Operational semantics of interface specifications under linear timing constraints:**

  Our work is the first to clearly define *operational* semantics of action diagrams under linear timing constraints.

- **Analysis of false negatives and false positives in known compatibility verification methods:**

  We show that known methods, e.g., [6], for the compatibility verification of timing diagrams under linear timing constraints can yield *false negative* answers to the compatibility question in practical situations. We also show that attempts to correct these known methods without taking the causality criterion into account can yield *false positive* answers to the compatibility question.

- **An accurate compatibility verification procedure:**

  We develop an accurate compatibility verification procedure for timing diagrams under linear timing constraints.

4 Thesis Organization and Overview

The thesis consists of seven chapters and three appendices. Chapters 2 to 6 are in the form of papers; the rest of this section provides logical bridges between these papers. Chapter 7 is the general conclusion of the thesis. Appendix I summarizes rules that must be followed when modeling with *Valid* and *Don't-care* valued actions. Appendix II is the grammar, in extended BNF form, of the HAAD language. Appendix III is the co-author's statements.

Chapter 2: A Stimulus/Response System Based on Hierarchical Timing Diagrams

In the course of validating system interfaces by simulation, the designer spends relatively large amounts of time writing "test benches" that perform stimulus generation and response validation (SGRV). We present a tool that facilitates this task by capturing the test bench specifications in the form of hierarchical action diagrams and modeling them using hierarchical constraint graphs. The specifications are then used to automatically perform SGRV. The main advantages of this approach are that many of the ad-hoc aspects of test bench creation are removed, thus contributing to the repeatability of the design validation process. Furthermore, since the overall control structure of the test bench and the correctness criteria that it uses to validate system responses, are captured declaratively (as opposed to detailed procedural code that interprets the specification), it follows that its test bench *intent* stands out more clearly.

A possible approach to the SGRV problem consists of generating all the stimuli before simulation, then performing the entire simulation, collecting traces of user specified signals, and, after simulation, validating circuit responses by pattern matching against the action diagrams. There are, however, several drawbacks to this "static" approach. First, it is incompatible with interactive simulation: for example, it does not support associating break-points with user specified error conditions. Second, the amount of data accumulated before and during simulation could become very large. The biggest drawback of such a static approach is, however, that it restricts the user from specifying stimuli that depend on the re-

sponse time of circuit outputs; for example, it is impossible to describe simple handshake protocols.

The alternative that we propose is dynamic SGRV (DSGRV) - i.e., an algorithm that traverses the constraint graph hierarchy during simulation to generate stimuli and validate system responses. We discuss a VHDL-based implementation of the tool and illustrate its usefulness and limitations in modeling microprocessor bus operations.

Chapter 3: Modeling and Execution of Timing Diagrams with Optional and Multi-Match Events

The algorithm of Chapter 2 requires every specified action to occur exactly once in a given execution of the enclosing timing diagram. However, the specification of certain types of timing constraints (e.g., set-up and hold times) in the context of an action-based model, requires actions (e.g., on a data bus) with symbolic values such as *Valid* and *Don't-care* that may or may not actually occur. Actions with such values cannot be handled by the DSGRV algorithm of Chapter 2. In this chapter, we introduce two new action types: ***optional actions*** (actions that do not always have to match actual action occurrences) and ***multi-match actions*** (actions that can match multiple actual action occurrences), and we consequently extend the execution model of Chapter 2.

Chapter 4: Integrating Behavior and Timing in Executable Specifications

In this chapter, we extend the set of action diagram composition operators of Chapter 2 to include *Choice* and *Loop* operators. We describe a general algorithmic framework in which it is easy to add new composition operators. In addition, we extend the specification paradigm to encompass the functional view of the specified system. For behaviors that have a control-flow which is governed primarily by the behavior's interface with the external world, this extension is done by allowing procedures and functions in the functional view to be "linked" to action triggers in the action view. For more general behaviors, the functional view is described by an EFSM (Extended Finite State Machine) which execution is synchronized to that of the system's action model. The synchronization is specified declaratively by the user.

We illustrate our approach on practical examples, and we show how we achieve tangible savings in model development time and accuracy.

Chapter 5: Modeling Cell Processing Hardware with Action Diagrams

In this chapter, timing constraints are explicitly classified into *assumptions* (i.e., assumptions on the environment of the described subsystem) and *commitments* (i.e., timing relations that the described subsystem commits to). The timing model is generalized to encompass both linear and non-linear timing constraints; this is done by defining three types of timing constraint composition operators: latest, earliest and conjunctive.

Furthermore, the action diagram composition operators are generalized as follows:

- The Choice composition operator is refined to support both deterministic and non-deterministic delayed choice semantics.
- A new operator is defined for exception handling.
- Port mappings, parameters, and local variables are added to action diagrams. The operator set now allows to build a true behavioral hierarchy.

Finally, features are defined to allow modeling at higher abstraction levels. These features include user-defined abstract data types and the distinction between message-based and value-based ports (in the former, actions are the results of an action diagram sending a message on the port, whereas in the latter actions are the result of value changes on the port). We illustrate the concepts on the high-level model of a cell-based (e.g., packet or ATM) communication sub-system.

Chapter 6: Semantics and Verification of Action Diagrams under Linear Timing Constraints

In this chapter, we examine the question of the compatibility of concurrent, communicating leaf action diagrams described by linear timing constraints. We show that known methods that address this question, e.g., [6], can yield *false negative* answers because they do not compose the interface behaviors of the communicating systems. We show that such composition must encompass the concept of realizability, or else the compatibility question can yield *false positive* answers.

We then formalize the operational semantics of action diagrams under linear timing constraints. The semantics are based on the derivation,

from the action diagram, of a *block machine* which is characterized by a partition of the action set of the action diagram. We define the concept of a *causal block machine* and we state the realizability of an action diagram specification in terms of the existence of a causal block machine derived from the action diagram. We prove that all causal block machines derived from an action diagram have the same (timed) trace set and this trace set is equal to that of the action diagram.

We define the compatibility of communicating causal action diagrams in terms of the compatibility of *all* the possible combinations of causal block machines derived from these action diagrams. We prove that we do not need to enumerate these combinations to answer the action diagram compatibility question. This leads to an exact and efficient procedure for the verification of the compatibility of communicating action diagrams.

Finally, we prove that the structure of the partition of the set of input actions of a causal block machine is independent of that of its output actions. In addition to being intuitively "reassuring", this property should be useful in designing an efficient action partitioning procedure.

References

[1] A. Silburt, Manager, Hardware Systems Modeling Group, Bell-Northern Research Ltd., Ontario, private communication, December 95.

[2] M. Meredith, Vice-President of Engineering, Chronology Corp., Washington, private communication, December 95.

[3] S. Curry, Cadence Design Systems Inc., private communication, December 95.

[4] I. Dobson, Director of Research & Development, Tundra Semiconductor Corporation, presentation at the Université de Montréal, February 96.

[5] A.R. Martello and S.P. Levitan "Causal timing verification", *1st ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1990.

[6] J.A. Brzozowski, T. Gahlinger and F. Mavaddat, "Consistency and Satisfiability of Waveform Timing Specifications", Networks, Vol. 21, 1991, pp91-107.

[7] K. McMillan and D. Dill, "Algorithms for Interface Timing Verification, Proc. ICCD-92, October 1992.

[8] T.M. Burks and K.A. Sakallah, "Min-Max Linear Programming and the Timing Analysis of Digital Circuits", Proc. ICCD-93, October 1993, pp152-155.

[9] P. Girodias, E. Cerny, W.J. Older, "Solving Linear, Min and Max Constraint Systems Using CLP Based on Relational Arithmetic," submitted to Int'l Conf. on Principles and Practice of Constraint Programming (CP95), Marseille, September 1995.

[10] S. Narayan, F. Vahid and D. Gajski, "System Specification and Synthesis with the SpecCharts Language", *IEEE Proc. ICCAD-91*, 1991.

[11] D. Drusinsky and D. Harel, "Using StateCharts for Hardware Description and Synthesis", in *IEEE Transactions on Computer-Aided Design*, 1989.

[12] P. Rony "Interfacing fundamentals: Timing diagram conventions", *Computer Design*, pp. 152–153, 1980.

[13] "Message Sequence Charts (MSC)", Recommendation Z.120, CCITT.

[14] G. Borriello, *A New Interface Specification Methodology and its Application to Transducer Synthesis*, PhD thesis, University of California, Berkeley, 1988.

[15] S.K. Sherman, "Algorithms for timing requirement analysis and generation", *ACM/IEEE Proc. 25th DAC*, pp. 724–727, 1988.

[16] F. Mavaddat and T. Gahlinger, "On deducing tight bounds from partial timing specifications", *1st ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1990.

[17] G. Borriello, "Combining event and data-flow graphs in behavioral synthesis", *IEEE Proc. ICCAD-88*, pp. 56–59, 1988.

[18] T. Amon, G. Borriello and C. Séquin, "Operation/event graphs: A design representation for timing behavior", *Computer Hardware Description Languages and their Applications*, IFIP, North-Holland, 1991.

[19] S.A. Hayati, A.C. Parker and J.J. Granacki, "Representation of control and timing behavior with applications to interface synthesis", *IEEE Proc. ICCD-88*, pp.382–387, 1988.

[20] M.C. McFarland, "CPA: Giving an account of timed system behavior", *1st ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1990.

[21] J.S. Ostroff, "Automatic verification of timed transition models", *Int'l Workshop on Automatic Verification Methods for Finite State Systems*, LNCS 407, Springer-Verlag 1989.

[22] J.S. Ostroff, "Real-time computer control of discrete event systems modeled by extended state machines: a temporal logic approach", Technical Report EE-86-18, University of Toronto, 1986.

[23] B. Berthomieu and M. Menasche, "An enumerative approach for analyzing Petri nets", *Information Processing 83*, Elsevier Science, North-Holland, 1983.

[24] R.H. Lathrop and R.S. Kirk, "An extensible object-oriented mixed-mode functional simulation system", *ACM/IEEE Proc. 22nd DAC*, pp. 630–636, 1985.

[25] M. Abramovici, D.T. Miller, J.J. Kulikowski, and P.R. Menon, "System-level design verification at the AT&T computer division: Tools", *IEEE Proc. ICCD-89*, pp. 548–554, 1989.

[26] A. Silburt, I. Perryman, J. Bergeron, S. Nichols, M. Dufresne and G. Ward, "Accelerating Concurrent Hardware Design with Behavioral Modeling and System Simulation" *ACM/IEEE Proc. 32nd DAC*, 1995.

[27] Y.H. Leong and W.P. Birmingham, "The Automatic Generation of Bus-Interface models", in *ACM/IEEE Proc. 29th DAC*, pp. 634–637, 1992.

[28] B.A. Gennart and D.C. Luckham, "Validating discrete event simulations using event pattern mappings", *ACM/IEEE Proc. 29th DAC*, pp. 414–419, 1992.

[29] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM, 1983.

[30] W. Older and A. Vellino, "Constraint Arithmetic on Real Intervals", *Constraints Logic Programming: Selected Research*, 1993.

[31] T. Amon, H. Hulgaard, G. Borriello, S. Burns, "Timing Analysis of Concurrent Systems: An Algorithm for Determining Time Separation of Events", Proc. ICCD-93, October 1993.

[32] M. A. Escalente and N. J. Dimopoulos, "Assessing the Feasibility of Hardware Interface Designs in Microprocessor-based Systems", Technical Report ECE-95-1, EE Dept., University of Victoria, 1995.

[33] D. C. Ku and G. De Micheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*, Kluwer Academic Publishers, 1992.

[34] A. S. Klusener, *Models and Axioms for a Fragment of Real-Time Process Algebra*, Ph.D. Thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1993.

[35] G.J. Milne, "The formal description and verification of hardware timing", *IEEE Trans. Computers, vol. 40, no. 7*, pp. 811–826, 1991.

[36] R. Cleaveland, J. Parrow, and B. Steffen, "The concurrency workbench: A semantics-based verification tool for finite-state systems", *Proc. Workshop on Automated Verification Methods for Finite-State Systems, LNCS 407, Springer-Verlag*, 1989.

[37] T. Yoneda, K. Nakade, and Y. Tohma, "A fast timing verification method based on the independence of units", *IEEE Proc. 19th FTCS*, pp. 134–141, 1989.

[38] D. Dill, "Timing assumptions and verification of finite-state concurrent systems", *Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, Springer-Verlag, 1989.

[39] R. Alur, C. Courcoubetis, D. Dill, "Model checking for real-time systems", *Proceedings of the fifth IEEE Symposium on Logic in Computer Science*, pp. 414–425, 1990.

[40] E. M. Clarke and E. A. Emerson, "Characterizing properties of parallel programs as fixpoints" *Seventh International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science 85, 1981.

[41] B. Berkane, S. Gandrabur, and E. Cerny, "Timing diagrams: semantics and timing analysis", *Proceedings of the Asian Pacific Conference on Computer Hardware Description Languages*, 1996.

[42] X. Nicolin et al., "From ATP to timed graphs and hybrid systems", *Acta Informatica, V30*, 1993.

[43] F. Jahanian and A.K.L. Mok, "A graph-theoretic approach for timing analysis and its implementation", *IEEE Trans. Computers*, C-36(8), pp. 961–975, 1987.

[44] F. Jahanian and A.K.L. Mok, "Safety analysis of timing properties in real-time systems", *IEEE Trans. Software Eng.*, vol. SE-12, no. 9, pp. 890–904, 1986.

[45] R. Groz, *Vérification de propriétés logiques des protocoles et systèmes répartis par observation de simulation*, Ph.D. thesis, Université de Rennes I, France, 1989.

[46] K. Khordoc, M. Dufresne, and E. Cerny, "A stimulus/response system based on hierarchical timing diagrams", *IEEE Proc. ICCAD-91*, pages 358–361, 1991.

[47] K. Khordoc, E. Cerny, and M. Dufresne, "Modeling and execution of timing diagrams with optional and multi-match events", *Proc. 2nd ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1992.

[48] Texas Instruments Incorporated, *Supplement to MOS Memory Data Book*, Texas Instruments, Houston, Texas, 1984.

[49] C.W. Moon, P.R. Stephan, and R.K. Brayton, "Synthesis of hazard-free asynchronous circuits from graphical specifications", *IEEE Proc. ICCAD-91*, pages 322–325, 1991.

[50] *IEEE Standard 1076-1987, VHDL Language Reference Manual*, IEEE, 1987.

[51] K. Khordoc, M. Dufresne, E. Cerny, P.A. Babkine and A. Silburt, "Integrating Behavior and Timing in Executable Specifications", in *IFIP Conference on Hardware Description Languages and their Applications (CHDL)*, 1993.

[52] K. Khordoc and E. Cerny, "Modeling Cell Processing Hardware with Action Diagrams", in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1994.

[53] K. Khordoc and E. Cerny, "Semantics and verification of action diagrams with linear timing constraints", submitted to *ACM Transactions on Design Automation of Electronic Systems*, 1995.

CHAPTER 2

A STIMULUS / RESPONSE SYSTEM BASED ON HIERARCHICAL TIMING DIAGRAMS

ABSTRACT

We present a tool that facilitates timing verification in the context of behavioral simulation. The tool captures timing specifications from hierarchical timing diagrams and models them using hierarchical constraint graphs. Our main contribution is a new algorithm that dynamically traverses the constraint graph hierarchy during simulation to generate stimuli and validate system responses. We discuss a VHDL-based implementation of the tool and illustrate its usefulness and limitations in modeling microprocessor bus operations.

1 Introduction

A number of static timing analysis tools [1, 2, 3] have been proposed for the verification of digital designs. These tools adequately address the realm of gate-level synchronous circuits. However, they are inadequate for large scale designs, where higher-level abstractions of timing properties must be used, in order to reduce the large amounts of data to be dealt with, or simply because such low-level data is not available (e.g., in the case of off-the-shelf VLSI components). Furthermore, with the increasing use of synthesis tools, the designer has less control over the gate-level implementation; therefore, any useful analysis tool must provide higher-level diagnostics which the designer can relate to. Finally, the design might be asynchronous at the system level (e.g., asynchronous bus interfaces), thus requiring other timing verification techniques.

As a consequence, there is presently a need for timing verification tools that address system level design. Although formal methods are beginning to emerge, e.g., extensions of predicate logic [4], graph based methods [5], timed automata [6] and concurrent process calculus [7], system level timing verification still relies on dynamic checking using behavioral simulation [8, 9, 10]. The problem with this approach is that designers spend a relatively large amount of time writing both the stimuli to drive the system inputs and the validation procedures to check whether the system verifies its timing and functional specifications.

In this paper, we propose a new tool that facilitates the timing verification of complex systems in the context of behavioral simulation. The tool captures timing specifications in the form of a hierarchy of formalized timing diagrams [11]. These diagrams are based on the concept of timing constraints, and thus they represent a set of allowable behaviors - the specifications - rather than one particular instance of behavior. Moreover, the diagrams resemble those supplied by component manufacturers and are well-understood by hardware designers. The proposed tool uses the timing specifications extracted from these diagrams to automatically perform Stimulus Generation and Response Validation (SGRV), relieving the designer from this tedious task.

A possible approach to the SGRV problem consists of generating all the stimuli before simulation, then performing the entire simulation, collecting traces of user specified signals, and, after simulation, validating circuit responses by pattern matching against the timing diagrams (TDs). There are, however, several drawbacks to this "static" approach. First, it is incompatible with interactive simulation: for example, it does not support

associating break-points with user specified error conditions. Second, the amount of data accumulated before and during simulation could become very large. The biggest drawback of such a static approach is, however, that it restricts the user from specifying stimuli that depend on the response time of circuit outputs; for example, it is impossible to describe simple handshake protocols.

In this paper, we consider the *dynamic* SGRV (DSGRV) problem, i.e., the problem of generating stimuli and verifying circuit responses *during* the simulation run-time. We propose a solution based on:
- capturing timing specifications using hierarchical timing diagrams,
- modeling timing specifications using a hierarchical extension to the constraint graph model [5, 12], and
- using the constraint graph hierarchy to stimulate the circuit and validate its responses dynamically.

Our main contribution is a new algorithm that dynamically traverses the constraint graph hierarchy during simulation to generate stimulus events and validate circuit responses. Although there are tools that perform static stimulus generation from a set of timing constraints [13], this is, to the best of our knowledge, the first published work that addresses the DSGRV problem. To demonstrate our ideas, we have implemented the DSGRV system as a VHDL [14] process that dynamically interacts with the simulated circuit.

The paper is structured as follows: Section 2 reviews related work. Section 3 presents our model and terminology. Sections 4 to 7 introduce the DSGRV algorithm in a gradual manner: static (Section 4) and dynamic (Sections 5 and 6) event generation, followed by response validation (Section 7) and hierarchical DSGRV (Section 8). Section 9 contains experimental results and discussions of the limitations of the system. Section 10 concludes the presentation.

2 Related Work

Timing constraints are typically supported by languages oriented towards the synthesis of interface circuits [11], however they are absent from stimulus/response description languages [15, 16, 17] which are oriented towards simulation and test.

Recently, a number of timing analysis systems [5, 12, 18, 19] have used a model in which hardware modules are represented by interface op-

erations consisting of a set of events interelated by timing constraints. These constraints are represented by a directed constraint graph, where nodes represent events, and a directed edge of weight a from node X to node Y represents the timing constraint: $t_Y - t_X \geq a$, with t_X and t_Y representing the occurrence times of events X and Y, respectively.

A problem of interest is the consistency of an interface operation, i.e., whether there exists an assignment of event times such that all constraints are satisfied. This problem can be solved [12, 18] by detecting cycles of positive weight in the graph. In [18], a constraint priority scheme defined by the user, is used to relax some constraints, thus removing the inconsistencies from the interface operation.

Another problem is the satisfiability of safety constraints by causality constraints, i.e., whether all possible time assignments that satisfy the causality constraints also satisfy the safety constraints. This problem is solved [12, 19] by comparing longest paths between pairs of events in the constraint graph. In [5] and [6], the satisfiability problem is solved under more expressive specification paradigms: first order logic and timed automata, respectively.

Finally, the problem considered in [13] is the generation of event times from a set of timing constraints specified as Prolog rules. The method uses the built-in backtracking mechanism of Prolog, however there is no provision for generating stimulus events in increasing time order, or in reaction to circuit responses.

In the next section, we introduce our model and the terminology used in the rest of the paper.

3 The Model

The timing specification of an interface operation consists of a set of signals and a set of timing constraints. Each signal is characterized by a name and a direction (input or output), and is composed of a totally ordered sequence of events. Bidirectional signals are modeled by separating their input and output components. An event value indicates the value of the corresponding signal after the occurrence of the event. Event values are in the set $V = B \cup \{s, u, z\}$, where B is the domain of the given signal subtype, e.g., $B = \{0, 1\}$ for a bit signal and $B = \{0, \ldots, 255\}$ for an 8-bit bus signal; s (*Stable*) represents any arbitrary value from B that does not change for a specified period of time, its actual value being irrelevant to the

timing specification; z stands for high-impedance, and u means *Don't-care* or *Unknown*. A timing constraint (or constraint for short) relates a pair of events. There are two types of constraints: *PREC* (precedence) and *CONC* (concurrency).

PREC represents event causality or event occurrence order in general, and is characterized by a minimum and a maximum time: *X PREC(min, max) Y* means that event Y must occur after event X by at least *min* units of time and at most *max* units of time. The *PREC* constraint can be expressed as:

$$\begin{aligned} & t_Y - t_X \geq min \\ \text{and} \quad & \\ & t_X - t_Y \geq -max \end{aligned} \tag{1}$$

where t_X and t_Y are variables that represent the occurrence times of events X and Y, respectively, and *min* and *max* are integers satisfying $0 \leq min \leq max$. The graph representation of (1) is in Fig. 1(a). The notation *X PREC(min) Y* is used when *max* is not specified (i.e., $max = \infty$); when *min* is not specified, the relation represents *X PREC(0, max) Y*.

CONC stands for concurrency: *X CONC(max) Y* means that the occurrence times of X and Y must be separated by at most *max* units of time. This is expressed as:

$$\begin{aligned} & t_Y - t_X \geq -max \\ \text{and} \quad & \\ & t_X - t_Y \geq -max \end{aligned} \tag{2}$$

where *max* is an integer such that $max \geq 0$. The graph representation of (2) is in Fig. 1(b).

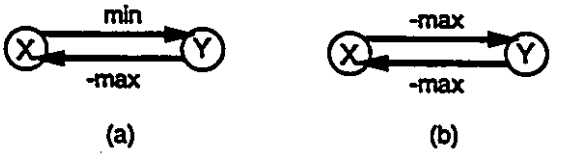


Figure 1: Graph representation of constraints. (a) PREC constraint. (b) CONC constraint.

In the rest of this paper, an event and the graph node representing it are used interchangeably. A "path" from event X to event Y means a directed path from X to Y in the constraint graph. Similarly a "cycle" in the graph stands for a directed cycle. The "weight" of a path (cycle) is the sum of the weights of the edges forming the path (cycle). The term

"positive path" ("negative path") stands for a path with weight strictly greater than zero (smaller or equal to zero). The notation $LP(XY)$ indicates the weight of the longest path, i.e., the path of maximum weight, from event X to event Y. The term "stimulus event" ("response event") indicates an event that the DSGRV system must generate (observe), i.e., an input (output) of the system being verified. The set of constraints is said to be consistent if there exists a time assignment to the events of the graphs such that all constraints are satisfied. It is well-known [5, 20] that this consistency property holds if and only if there are no positive cycles in the constraint graph (CG). A positive cycle can be detected as a side effect of the longest paths computation [21].

When the constraint graph is constructed, an "Origin" event O is created to represent time 0, and a *PREC(0)* relation is added from O to the first event of every signal, and between any two successive events of the same signal if no other *PREC* relation was specified between them. Furthermore, an *End* event is added with a *PREC(0)* relation from the last event of every signal to the *End* event.

4 Static Generation of Stimuli

In this section, we consider the problem of fixing event times in a static fashion, i.e., outside the simulation context, such that all timing constraints are satisfied. Only stimulus events are considered in this static context. As events are assigned occurrence times, the constraint system is modified; indeed, fixing the occurrence time of an event X to a time t_X is equivalent to adding edges of weight t_X and $-t_X$ from events O to X and X to O, respectively. In the following, a "free event" designates an event which has not yet been assigned an occurrence time. The event becomes a "fixed event" once it is assigned an occurrence time.

Lemma 1: Given a consistent constraint system and a free event X picked arbitrarily from the set of free events, the constraint system remains consistent when fixing X iff t_X is chosen such that: $LP(OX) \leq t_X \leq -LP(XO)$.

Proof: Let $LP(OX)$ and $LP(XO)$ be the weights of the longest paths from event O to event X and from event X to event O, respectively (Fig. 2). Note that the two paths form a cycle of negative weight due to the consistency of the system prior to fixing X. Therefore, the following holds:

$$LP(OX) \leq -LP(XO) \tag{3}$$

If we now fix X at some time t_X, we add the two edges of weight t_X and $-t_X$ (Fig. 2). The maximum weight cycle involving the $-t_X$ edge is of weight: $LP(OX) - t_X$; it is of maximum weight because it involves the longest path from O to X. To maintain consistency of the system, the weight of this cycle must be negative; therefore:

$$t_X \geq LP(OX) \tag{4}$$

Similarly, the maximum weight cycle involving the t_X edge is of weight $t_X + LP(XO)$. To maintain consistency of the system, this weight must be smaller or equal to 0; therefore:

$$t_X \leq -LP(XO) \tag{5}$$

It follows from (4) and (5) that $LP(OX) \leq t_X \leq -LP(XO)$. Note that this interval is non-empty due to (3). Similarly, it can be easily shown that any value of t_X outside this interval creates a positive cycle and thus an inconsistent system.

Q.E.D.

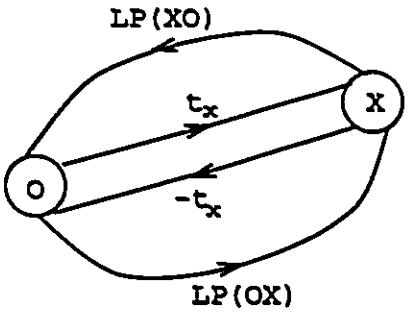


Figure 2: Occurrence interval of event X.

In the following, the time interval of Lemma 1 is designated as the "occurrence interval" of the event.

5 Dynamic Generation of Stimuli

In this section, we consider the problem of dynamically fixing event occurrence times during simulation. We assume an HDL based behavioral simulation environment with support for concurrent processes, such as found in VHDL [14]. Processes have their own internal variables and

data-structures. Inter-process communication is performed using signals: a process reads the values of its input signals and drives values on its output signals with a delay greater or equal to zero which may result in the scheduling of events. There is one global simulation time clock in the system; processes execute with simulation time frozen and return control to a global scheduler using the "WAIT" synchronization primitive. WAITs have resume conditions associated to them such as timeouts, specific event occurrences on signals, or combinations of these. The scheduler advances simulation time depending on the temporal latency of the system and gives control back to the processes for which the WAIT conditions have become true.

In this model, we view the DSGRV as a process which communicates with the circuit being simulated via a set of I/O signals. A question then arises as to the synchronization of the DSGRV process with the system under verification. There are three basic options: 1- one extreme is to schedule all events in the future when the DSGRV process takes control; 2- the other extreme is to fix the occurrence instant of at most one event every time the DSGRV process resumes its execution; then, when the current time reaches that occurrence instant, drive the corresponding signal with zero delay; finally 3- is some intermediate solution whereby each time the DSGRV process takes control, it schedules groups of events in the future. Obviously, option 1 does not meet the requirements for DSGRV as outlined in Section 1. Furthermore, in order to simplify the DSGRV algorithm, we choose option 2 over option 3.

In order to generate events dynamically, requirements additional to those presented in Section 4 must be placed on the generation process (Rules 1 to 3 below). We use the term "past event" instead of "fixed event" and "future event" instead of "free event", to indicate the existence of a forward running time clock.

Rule 1: If two future events X and Y are such that $LP(XY) > 0$, then X must be generated before Y.

In the following, a future event X is "feasible" if for all future events $Y \neq X : LP(YX) \leq 0$. Note that given a non-empty set S of future events, it is always possible to find a feasible event in S, otherwise there would be a positive cycle in the constraint graph.

Rule 2: Given two future events X and Y, where X is a feasible event (i.e., $LP(YX) \leq 0$), the occurrence time of X must be chosen such that no positive path is created from Y to X, i.e., the inequality $LP(YX) \leq 0$ must be preserved.

Informally, Rule 2 means that the occurrence time of X must not be chosen in the future of Y. This rule can be applied using the following lemma.

Lemma 2: For two future events X and Y, such that $LP(YX) \leq 0$, the relation $LP(YX) \leq 0$ is preserved upon the occurrence of X iff the occurrence time t_X of X satisfies: $t_X \leq -LP(YO)$.

Proof: The occurrence of X at t_X creates a new Y to X path of weight $LP(YO) + t_X$ (Fig. 3). In order to prevent this path from being positive, we must have: $LP(YO) + t_X \leq 0$, i.e., $t_X \leq -LP(YO)$. Note that it is always possible to satisfy Lemma 1 and the bound $t_X \leq -LP(YO)$. This is because these two bounds are conflicting only when $-LP(YO) < LP(OX)$, i.e., $LP(YO) + LP(OX) > 0$. Since $LP(YX) \geq LP(YO)+LP(OX)$, it follows that $LP(YX) > 0$, which is in contradiction with the assumption $LP(YX) \leq 0$.

Q.E.D.

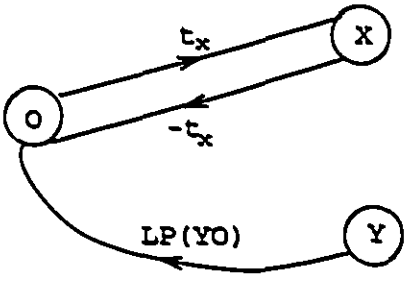


Figure 3: Occurrence of X creates new Y to X path.

Rule 3: The occurrence time t_Y of a future event Y must be such that: $t_Y \geq t_X$, where t_X is the occurrence time of a past event.

In the following lemma we show that it is always possible to find a time t_Y in the occurrence interval of Y, such that Rule 3 is respected.

Lemma 3: Given a future event Y and a past event X, it is always possible to make Y occur at some time t_Y that satisfies the bounds of both Rule 3 ($t_Y \geq t_X$) and Lemma 1 ($LP(OY) \leq t_Y \leq -LP(YO)$).

Proof: Rule 3 contradicts Lemma 1 only when $-LP(YO) < t_X$, i.e., $-LP(YO) < LP(OX)$ (because $LP(OX) = t_X$ after the occurrence of X). This yields $LP(YO)+LP(OX) > 0$. Since $LP(YX) \geq LP(YO)+LP(OX)$, it follows that $LP(YX) > 0$. This is a contradiction because:
1- $LP(YX)$ was ≤ 0 before the occurrence of X (otherwise X would not have been chosen to occur before Y), and,

2- $LP(YX)$ is guaranteed to stay ≤ 0 after the occurrence of X, due to Rule 2 and Lemma 2.

Q.E.D.

The following theorem specifies the allowed time interval for the dynamic generation of an event. This interval is designated as the "feasibility interval" of the event.

Theorem 1: Let $\{P_j\}$ and $\{F_i\}$ be the set of past and future events, respectively. An event $X \in \{F_i\}$ may be dynamically generated with the constraint system remaining consistent, iff X is chosen such that $\forall F_i \in \{F_i\}$, $LP(F_iX) \leq 0$ and the occurrence time t_X of X is chosen such that: $\max(LP(OX), \max_j(t_{P_j})) \leq t_X \leq \min(-LP(XO), \min_i(-LP(F_iO)))$

Proof: It follows directly from Rule 1, Lemmas 1 to 3, and the fact that it is impossible that a future event F_i is such that: $-LP(F_iO) < t_{P_j}$, where P_j is a past event (the proof is exactly the same as in Lemma 3, with P_j and F_i being respectively the X and Y events of Lemma 3).

Q.E.D.

6 Improved Dynamic Generation

The approach suggested in the previous section for the solution of the dynamic generation problem is relatively inefficient due to the computation of longest paths between all pairs of future events, required by Rule 1. In this section we propose two improvements that allow the reduction of the number of future events to be considered in the application of this rule.

Consider the graph CG* obtained by ignoring the CONC constraints and by representing each *PREC* constraint of the form *X PREC(min, max) Y* as an (unweighted) directed edge from X to Y. The resulting graph is acyclic (otherwise the original graph CG would contain a positive cycle). Furthermore, CG* connects all events of CG and there is a directed path in CG* from the Origin to every event of CG, because by construction there is a *PREC* relation between the Origin and the first event of each signal, and between any two consecutive events of the same signal. The *PREC* relation thus defines a topological sort on the event set. In the following, we say that X is a "predecessor" of Y (or Y is a "successor" of X), if X *PREC* Y; similarly, we say that X is an "ancestor" of Y if there exists $Z_1 \ldots Z_n$ such that X *PREC* Z_1, Z_i *PREC* Z_{i+1} for $i = 1 \ldots n$, and Z_n *PREC* Y. Furthermore, the term "frontier" designates the subset of future events for

which all predecessors have occurred.

The first improvement limits the computation of longest paths to the pairs of elements of the frontier. Initially, the frontier contains the Origin event only; it is then updated incrementally by traversing CG* in a PERT fashion [22], i.e., by inserting an event X in the frontier when all its predecessors have occurred and removing X when it occurs. Using Rule 1 of Section 5 and the fact that X *PREC* Y implies $LP(XY) \geq 0$, we deduce that a necessary condition for an event to be feasible is that all its predecessors have occurred; thus the set of feasible events, designated as the "feasible set", is a subset of the frontier. However, membership in the frontier is not a sufficient condition for an event to be feasible. For example, in Fig. 4, assume event V has just occurred, and all other events in the figure are future events. Event X is in the frontier, since all its predecessors have occurred. However, X is not feasible, because there exists a positive path (of weight 10) from future event Y to X (note that Y is not even in the frontier). In the previous section, the order of occurrence "Y before X" was established by examining the longest paths between all pairs of future events. Lemma 4 below assures that, even when we restrict the LP computations to pairs of elements of the frontier, we cannot inadvertently "forget" such Y events, and thus the correct order of event occurrences is preserved.

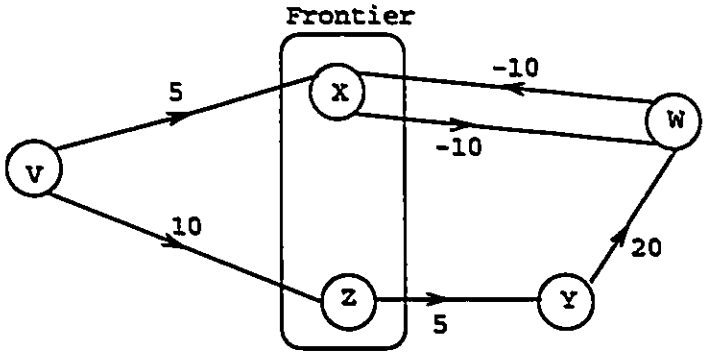


Figure 4: Frontier event X is not feasible.

Lemma 4: If for some event X in the frontier there exists a future event Y, such that $LP(YX) > 0$, then either Y is in the frontier, or some ancestor Z of Y is in the frontier such that $LP(ZX) > 0$.

Proof: It stems from the conjunction of the following two facts:
1- The *PREC* relation defines a partial order on the events. Furthermore, due to the connectivity property of CG* and the manner in which the frontier is built, the frontier is a maximal unordered set (by the *PREC*

relation). Therefore any future event Y which is not in the frontier must be ordered by the *PREC* relation with respect to some element Z in the frontier. Since Y has not occurred yet, it cannot precede Z, hence it must follow Z. As a result, any future event Y which is not in the frontier must have an ancestor Z in the frontier.
2- Since $LP(ZY) > 0$ (because Z is an ancestor of Y) and $LP(YX) > 0$ (by assumption), then $LP(ZX) > 0$ (because $LP(ZX) \geq LP(ZY) + LP(YX)$).

Q.E.D.

Note that the size of the frontier is bounded above by the number of signals in the timing specification. Since this number is in general much smaller than the number of future events, the pairs longest paths computation on the frontier proceeds substantially faster than on the complete set of future events.

Let $\{G_k\}$ designate the frontier. The second improvement to the dynamic generation method consists of eliminating from the feasible set any frontier event G_l such that $LP(OG_l) > \min_k(-LP(G_kO))$. This is because if there exists an event $G_k \in \{G_k\}$ such that $LP(OG_l) > -LP(G_kO)$, then $LP(G_kG_l) > 0$ (the proof is trivial). For an event G_l so eliminated, we do not need to compute $LP(G_kG_l)$, $\forall G_k \in \{G_k\}$.

7 Observation of Responses

In this section, we add the validation of system responses to the dynamic generation process and present the complete DSGRV algorithm (Fig. 5). In addition to timeouts for generating stimulus events, the WAIT condition in this algorithm considers response event activity and timeouts for the absence of expected response events. There are two types of response errors. The first type is signaled by the procedure *match-events* and is due to the occurrence of a response event that does not match any event in the list *expected-R-events* (this list is the expected response events subset of the feasible set). Matching is based on signal name, event value and occurrence time, i.e., the occurrence time of a response event must be in the occurrence interval (Lemma 1) of the corresponding expected event. The second type of response error is signaled by the procedure *response-time-out-error* when the DSGRV process is woken up due to a response timeout (i.e., T = *max-R-time*) and no response event has occurred; a response timeout error is then indicated for all events with expired time intervals. Note that *max-R-time* is the smallest of the upper bounds of the

occurrence time intervals in *expected-R-events*.

The procedure *update* updates the frontier and the longest paths, and it computes the feasible set and *maxtime*, where *maxtime* is used in the computation of feasibility intervals and is equal to $\min_k(-LP(G_kO))$, $G_k \in$ *Frontier*. The notation *inf(X)* and *sup(X)* indicates the quantities $LP(OX)$ and $-LP(XO)$, respectively. The Origin and End nodes are treated as pseudo stimulus events (in the sense that they are assigned occurrence times); however their generation does not produce any simulator event activity.

At this point, it is useful to note that DSGRV is intended to be a relatively low-level utility on which "intelligent" simulation-based services can be built. For example, choosing event occurrence times to test "marginal" or "average" conditions is an interesting problem. The interface of the DSGRV with the analysis tool that addresses this problem can be done through the function *choose-from-interval*. In our current prototype implementation of DSGRV, we simply choose the mid-point of (closed) intervals. In case of semi-infinite intervals (i.e. when $LP(XO) = -\infty$), we choose a constant offset from the interval's lower bound. Random choice is another possibility.

8 Hierarchical Timing Diagrams

The objectives of a hierarchical specification of timing diagrams are twofold: 1- facilitate the re-use of previously defined TDs when defining more complex specifications, and 2- minimize the computation time needed for longest paths updates in complex timing diagrams.

We have defined two basic TD composition operations: horizontal, i.e. concatenation along the time axis (*TDConcat*) and vertical, i.e. putting TDs in concurrency (*TDConcur*). These composition operations are expressed in terms of hierarchical graphs (*hgraphs*), as shown in Fig. 6. In this model, the direct subgraphs (e.g. the Q_i's in Fig. 6) of a given graph (e.g. P in Fig. 6) are represented by their Origin and End event nodes. Constraints can be placed between Origin/End nodes of Q_i's and P; constraint edges are allowed to "traverse" a TD "boundary" only at its Origin and End nodes. The value of a signal before its first event in a TD is taken to be equal to the signal value after its last event in a previous TD, or unknown if such a previous event does not exist.

In *TDConcat* (Fig. 6(a)), any event in Q_i occurs after all events in

Q_{i-1} have occurred; more specifically, O_i, the Origin event of a given Q_i is generated with a delay between a_i and b_i when E_{i-1}, the End event of Q_{i-1} occurs (or, in the case of $i = 1$, after O, the Origin of P, occurs).

In *TDConcur* (Fig. 6(b)), we note that:

$$\forall Q_i \in P, \forall Q_j \in P : LP(O_i, O_j) = LP(O_j, O_i) = -c$$

where c is the concurrency time of *TDConcur*; this implies that: O_i *CONC(c)* O_j. Furthermore, O_i, the Origin event of a given Q_i is generated with a delay between 0 and c when O, the Origin event of P, occurs. Note that, for a *TDConcur* composition to be meaningful, the Q_i's must be defined over disjoint sets of signals.

The advantage of the *hgraph* model is that it offers unified representation and processing of leaf and composite TDs. However, such a model taken in its full generality, would be unable to limit the ripple effects of longest paths updates to within the graph where a given event occurs and thus, would not achieve the efficiency objective stated at the beginning of this section. Instead, we take advantage of the special characteristics of *TDconcat* and *TDConcur* to define *hgraph**, a restricted *hgraph* model for which an efficient LP update algorithm can be defined. This is done while at the same time, conserving the advantage of a unified representation and processing of leaf and composite TDs. Furthermore, the *hgraph** model is general enough to support the definition of new TD composition operations.

In the following, Q_i is a direct subgraph of a graph P, O and E designate the Origin and End events of P, and O_i and E_i designate the Origin and End node of Q_i. Furthermore the notation $O\vec{O}_i$ designates a path from O to O_i and $w(O\vec{O}_i)$ designates the weight of this path. For the purpose of characterizing the *hgraph** model, we consider a *reduced* hierarchical graph in which each *leaf* graph Q_j is represented by its O_j and E_j nodes and by the two arcs O_jE_j and E_jO_j of weight $LP(O_jE_j)$ and $LP(E_jO_j)$ respectively, where these longest paths are computed strictly inside Q_j. The characteristics of a graph Q_i in the *hgraph** model are as follows:

1. Only PREC relations are used if Q_i is a non-leaf graph (i.e. no CONC relations are allowed; note however that TDConcur *is* allowed at any level of the hierarchy).

2. $\exists O\vec{O}_i$. $\forall O\vec{O}_i$, $w(O\vec{O}_i) \geq 0$ and $O\vec{O}_i$ is strictly *outside* Q_i.

3. $\exists O_i\vec{E}_i$. $\forall O_i\vec{E}_i$, $w(O_i\vec{E}_i) \geq 0$ and $O_i\vec{E}_i$ is strictly *inside* Q_i.

4. $\forall O\vec{E}_i$, $O\vec{E}_i$ is a concatenation of a $O\vec{O}_i$ and a $O_i\vec{E}_i$, where these two paths are according to the above two characteristics.

5. $\forall E_i\vec{O}_i$, if such (a) path(s) exist(s), $E_i\vec{O}_i$ is strictly *inside* Q_i.

6. $\exists E_i\vec{E}$. $\forall E_i\vec{E}$, $w(E_i\vec{E}) \geq 0$.

The above characteristics imply that for any event X in a *non-leaf* Q_i, LP(OX) and LP(XO) depend strictly on events of P which are in the past of O_i, and on events of Q_i which are in the past of X (in the case of a *leaf-level* Q_i, these longest paths can also depend on events of Q_i which are in the future of X, however they do not depend on future events outside Q_i).

Another consequence of the above characteristics is that, for any pair of *frontier* events X_1 in Q_1 and X_2 in Q_2, $LP(X_1X_2) = LP(X_1O) + LP(OX_2)$. This is because there are no $X_1\vec{X}_2$ paths that pass through the future of X_1 or X_2 and because all predecessors of X_1 and X_2 have occurred (since X_1 and X_2 are frontier events). As a result, longest paths between pairs of frontier events of different subgraphs need not be computed in determining feasible sets. Instead, to determine if there is a positive path between X_1 and X_2, it is sufficient to compare $LP(X_1O)$ to $LP(OX_2)$.

Using bottom-up recurrence across the hierarchy, it can be easily proved that the longest paths properties of the non-leaf graph Q_i can be extended to LP(O'X) and LP(XO') where O' is the Origin of any graph which recursively contains Q_i. The consequence of this is that when event X occurs, longest paths update operations need be performed only in Q_i. Actually, the update operation can be limited to the events in the (updated) frontier of this graph and can be performed in O(1) time (assuming a "constant" fan-out degree of event nodes).

The hierarchical DSGRV process starts at the root TD and progressively advances its frontier, recursively opening lower level TDs in a top-down fashion, as the Origin events of these TDs occur, then closing the TDs in a bottom-up fashion, as their End events occur. Each TD in the hierarchy stores its frontier, feasible set, maxtime, and list of *active children*. A child of a TD is said to be *active* if its Origin event has already occurred, and its End event has not occurred yet. The frontier of a TD contains events that are strictly local to the TD, while the feasible set and maxtime of the TD are cumulative for the whole subtree headed by the TD.

When an event occurs, update operations are performed by the procedure *h-update* (Fig. 7), i.e. the call to *update(event, TD)* in Fig. 5 is replaced by the call to *h-update(event, TDPath(event))*. *TDPAth* is an ordered list of TDs, from the root to the "owner" TD of the event, where the owner is the TD that had requested execution of the event. The update operation consists first of locally updating the LPs and the frontier in the owner TD. Note that the procedure *update-frontier-and-LPs* has different methods of longest path computation in leaf TDs and composite TDs (since in leaf TDs the longest paths can depend on events of the TD which are in the future of the TD's frontier).

Then, if the event is the Origin of a child TD, the parent TD removes the event from its frontier, puts the corresponding child TD (*triggeredTD(event)*) on its active list of children and initiates the child TD for execution. If on the other hand, the event that occurred is the End event of a TD, the TD removes its End node from its local frontier. The parent TD recognizes that one of its children has terminated execution when it receives an empty feasible set from that child TD; the parent TD then removes the terminated child TD from its active list of children and puts the successor(s) of the occurred End event in its frontier. For convenience, an Origin (resp. End) event is represented by two distinct objects in both the TD it starts (resp. ends) and in the parent TD; the accessor *associated-event(event)* allows to pass from one representation to the other.

Next, feasible set update operations (implemented by *h-compute-feasible-set*, Fig. 8) are performed bottom-up from the owner TD; at each ancestor of that TD, the feasible set is combined with the feasible set of other branches of that ancestor, and with the local frontier of the ancestor. Note that pairs longest paths are computed only in the case of leaf TDs. The feasible set that results at the root TD is then used for stimulus generation and response validation, as in the "flat" algorithm.

Finally, note that all event times and longest paths are with respect to the Origin of the root TD; this is done by initializing inf(Origin) and sup(Origin) of a given TD to the actual occurrence time of the Origin, instead of zero as was done in the "flat" case.

9 Experimental Results

We have implemented the DSGRV algorithm in VHDL and run experiments using different microprocessor bus operations. One series of examples we present here is the simulation of READ transactions between an

Intel 8085 CPU [23] (Fig. 9 and Table 1) and an 8355 ROM WITH I/O [23] (Fig. 10 and Table 2). The experiments are performed on a SUN 3/260 running Intermetrics VHDL.

Each chip is modeled as a separate DSGRV process using the specifications of Figures 9 and 10 and Tables 1 and 2. The READ operation is performed without wait states (i.e. the READY signal is not used). The IOW input line of the 8355 is not modeled, since it is used only for the I/O section of the chip. The CE (chip enable) bit of the ROM is driven by bit A_{11} of the CPU Address bus. Bits A_{12} to A_{15} of the CPU Address bus are left unconnected (they are outside the address range of the ROM). The rest of the Address bus and the AD (multiplexed Address/Data bus) are modeled as integer types (i.e. one "line" carrying an integer value for each bus). The other connections between the CPU and the ROM are evident from the corresponding port names. The input events that correspond to the beginning of expected valid Address/Data windows (windows labeled ADDRESS, DATA and DATA IN in Figures 9 and 10) are assigned the symbolic value *Stable*. The corresponding output Data or Address transitions generated by the DSGRV process on the driving side, are replaced by constants (address to be read and data content at this address); these constants are passed as arguments to the TDs at instantiation time of the TD hierarchy.

In the first experiment, a single READ operation is performed. Each DSGRV process contains a single "flat" graph describing the corresponding READ operation (Figures 9 and 10). The event activity resulting from the simulation is given in Table 3.

In the second experiment, two consecutive READ operations are performed. Each DSGRV contains a "flat" graph consisting of a "flat" concatenation of two READ operations.

In the third experiment, the same two consecutive READ operations are performed. However this time, each DSGRV contains an *hgraph* consisting of a *TDConcat* composition of two READ operations. It is interesting to note that the CPU specification is such that there are events near the end of the READ operation that impose constraints on events at the beginning of the next READ. For example, the rising edge of ALE (Address Latch Enable) at the beginning of a READ operation must be after the rising edge of the RD signal in the previous READ operation by a minimum of t_{CL} (50 ns). Similarly, the stable Address transition at the beginning of a READ must be after the rising edge of the RD signal in the previous READ operation by a minimum of t_{CA} (120 ns). Such constraints cannot be represented since, in the hierarchical graph model,

all inter-TD constraints must pass through Origin or End nodes of TDs. The specification of the CPU was therefore slightly modified as follows: A separation of a minimum of t_{CL} (50 ns) was specified between the two concatenated READ operations and an edge of weight $t_{CA} - t_{CL}$ (70 ns) was added from the Origin event of the READ TD to the Address Stable transition. Further investigation is needed to determine how to handle such cases without modifying the specifications nor flattening the hierarchy (which would yield slower DSGRV run-times).

The number of nodes and edges used in each DSGRV process and the run-time of the experiments are shown in Table 4. The columns labeled *Nodes* and *Edges* in this table account for all graph nodes and edges, respectively, in the corresponding DSGRV process, including those attributed to the different Origin/End nodes. The column labeled *events/sec* shows the "true" performance of the simulation, i.e. it accounts only for the events which generate simulation activity (this therefore excludes Origin/End "events"). We can see from Table 4 that in the case of "flat" specifications, the number of events processed per second decreases by the same factor as that of the increase in the size of the TD. This is of course due to the longest path algorithm in leaf TDs. However, in the case of hierarchical specifications, the performance of the DSGRV process in number of processed events per second, is practically independent of the size of the data set.

We conclude this section by outlining other limitations that we have encountered while using the DSGRV system.

- The DSGRV system is presently unable to model OR-type constraints (i.e. earliest firing events). For example, it is not uncommon to see the output bus drivers of DRAM chips be controlled by two signals such that, as soon as one of the two control signals is disabled, it turns the drivers off. Handling such cases requires a generalization of the longest paths algorithm. As for frontier updates, the rule for early firing events would be to put the event in the frontier as soon as one of its predecessors occurs.

- Another limitation of the DSGRV system is the absence of conditionals and states in the semantics of the model. Although we have implemented a TDChoice composition operation, the branching mechanism is under sole control of the user (through a user-written VHDL procedure which does the actual choice during the simulation runtime, with the help of the standard VHDL signal predicates). In order to improve the functionality of TDChoice, the system must

be able to automatically "match" multiple TDs in parallel, against observed events, and progressively eliminate those TDs which do not match the observed activity (such functionality is needed for example in the case of a memory component which must "decide" which operation the CPU is requesting). This functionality is, however, quite simple to integrate with the DSGRV algorithm presented here.

10 Conclusion

We have presented a novel approach to stimulus generation and response validation, based on timing constraint graphs. The merit of our approach is that it allows the stimulus/response system to dynamically interact with the circuit during the simulation run-time, thus allowing the generation of stimuli that depend on the response time of circuit outputs. We have extended the stimulus/response system to hierarchical constraint graphs and shown that this extension improves the simulation run-time at the expense of some loss in power of expression, namely that constraints that cross hierarchical boundaries cannot be expressed without some modifications to the original specifications. Finally, we have identified areas of future work such as the extension of the model to include conditional execution semantics and early firing events.

References

[1] R.B Hitchcock. Timing verification and the timing analysis problem. In *ACM/IEEE Proc. 19th DAC*, pages 594–604, 1982.

[2] T.G Szymanski. LEADOUT: A static timing analyzer for MOS circuits. In *IEEE Proc. ICCAD-86*, pages 130–133, 1986.

[3] M.R Dagenais and N.C Rumin. On the calculation of optimal clocking parameters in synchronous circuits with level-sensitive latches. *IEEE Transactions on CAD*, 8(3):268–278, March 1989.

[4] G.V Bochman. Hardware verification with temporal logic: An example. *IEEE Transactions on Computers*, C-31(3):223 231, March 1982.

[5] F. Jahanian and A.K.L Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8), August 1987.

[6] K. Nakade, T. Yoneda, and Y. Tohma. A fast timing verification method based on the independence of units. In *IEEE Proc. 19th FTCS*, pages 134–141, 1989.

[7] G.J Milne. Timing constraints: Formalizing their description and verification. In *Proc. 9th IFIP Symposium on CHDLs*, pages 103–116, 1989.

[8] R.H Lathrop and R.S. Kirk. An extensible object-oriented mixed-mode functional simulation system. In *ACM/IEEE Proc. 22nd DAC*, pages 630–636, 1985.

[9] Y. Huh, D.C Luckham, L.M Augustin, B.A Gennart, and A.G Stanculescu. Verification of VHDL designs using VAL. In *ACM/IEEE Proc. 25th DAC*, pages 48–53, 1988.

[10] D.T Miller, M. Abramovici, J.J Kulikowski, and P.R Menon. System-level design verification at the AT&T computer division: Tools. In *IEEE Proc. ICCD-89*, pages 548–554, 1989.

[11] G. Borriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis.* PhD thesis, University of California, Berkeley, 1988.

[12] T. Gahlinger, J.A Brzozowski, and F. Mavaddat. *Consistency and satisfiability of waveform timing specifications.* Research Report CS-88-24, University of Waterloo, 1988.

[13] R. Rastogi, A. Kara, and K. Kawamura. TDS: An expert system to automate timing design for interfacing VLSI chips in microcomputer systems. In *IEEE Proc. ICCAD-86*, pages 362–365, 1986.

[14] IEEE. *IEEE Standard 1076-1987, VHDL Language Reference Manual.* IEEE, 1987.

[15] R. Mathews I.M Watson, J.A Newkirk and D.B Boyle. ICTEST: A unified system for functional testing and simulation of digital IC's. In *IEEE Proc. ITC-82*, pages 499–502, 1982.

[16] J. Ivie and K. Lai. STL: A high-level language for simulation and test. In *Proc. 23rd DAC*, pages 517–523, 1986.

[17] A. Gilman. Logic modeling in WAVES. *IEEE Design and Test of Computers*, pages 49–55, June 1990.

[18] S.K Sherman. Algorithms for timing requirement analysis and generation. In *ACM/IEEE Proc. 25th DAC*, pages 724–727, 1988.

[19] S.P Levitan, A.R Martello, and D.M Chiarulli. Timing verification using HDTV. In *ACM/IEEE Proc. 27th DAC*, pages 118–123, 1990.

[20] E.L Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, New York, 1976.

[21] R.E Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, 1983.

[22] J.J Moder and C.R Phillips. *Project Management with CPM and PERT.* Van Nostrand, New York, 1970.

[23] Intel Corporation. *MCS-85 User's Manual.* Intel, Santa Clara, CA, 1978.

```
PROCESS DSGRV(TD, stimulus_signals, response_signals)
  S_event := origin(TD); S_time := 0; feasible_set := {S_event};
  expected_R_events := NULL; max_R_time := +infinity;

  while feasible_set not empty do
    T := get_current_simulation_time();

    /* GENERATE */
    if ((T = S_time) and (S_event =/ NULL))
      then occur_now(S_event); /* make X occur with 0 delay */
    end if;

    /* OBSERVE & VALIDATE */
    actual_R_events := get_actual_R_events(); /* query sim. */
    if actual_R_events
      then
        occurred_R_events :=
              match_events(actual_R_events, expected_R_events);
    elseif (T = max_R_time)
      then
        response_time_out_error(expected_R_events);
    end if;

    /* UPDATE */
    loop for event in {S_event} U occurred_R_events
         do
         (feasible_set, maxtime) := update(event, TD);
    end loop;
    expected_R_events := get_response_events(feasible_set);
    max_R_time := min_over_[X in expected_R_events](sup(X));

    /* CHOOSE */
    S_event := choose_stimulus_event(feasible_set);
    S_time := choose_from_interval[max(inf(S_event), T),
                                   min(sup(S_event), maxtime)];

    /* WAIT */
    timeout := (min (max_R_time, S_time)) - T;
    wait on response_signals for timeout;
  end while;
end DSGRV.
```

Figure 5: DSGRV Algorithm.

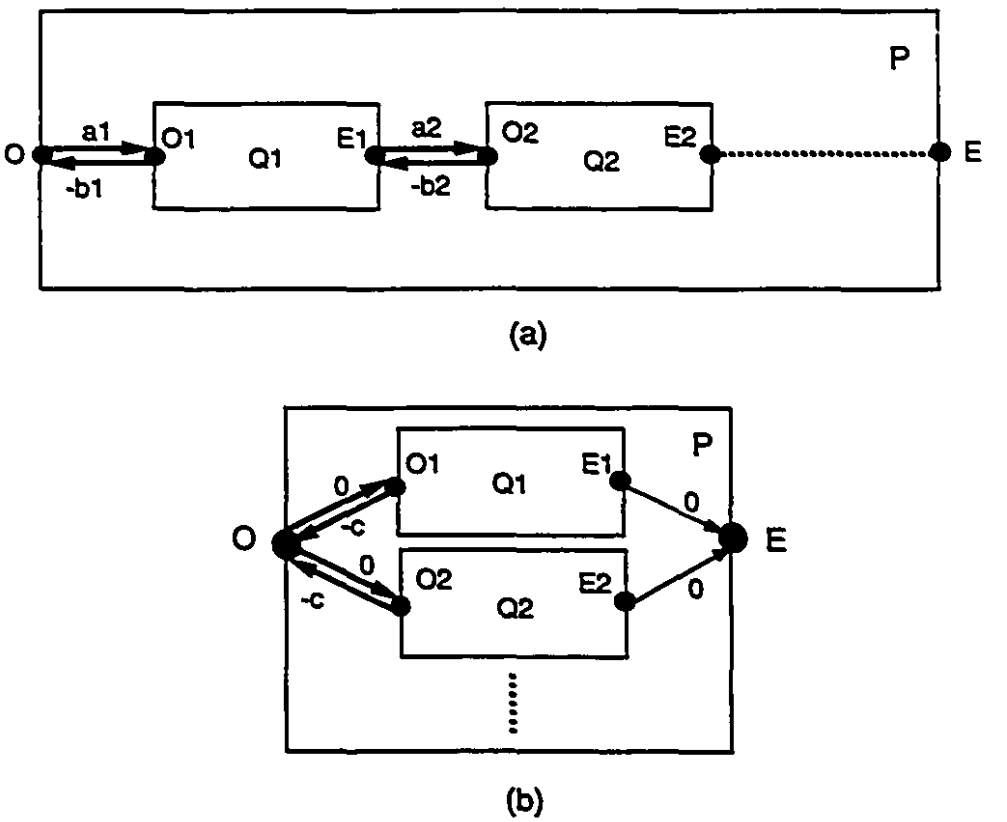


Figure 6: TD composition. (a) TDConcat composition. (b) TDConcur composition

```
PROCEDURE h_update(event, TDPath)
  TD := first(TDPath);
  if (rest(TDPath))    /* event belongs to a lower level TD */
    then               /* recursive call on child TD */
      (childFeasible_set, ChildMaxtime) :=
                          h_update(event, rest(TDPath));
      if (empty(childFeasible_set))
        /* event was the End event of the child TD */
        then
          /* remove the child TD from
                      the active children list of this TD */
          activeChildren(TD) :=
                   activeChildren(TD) - first(rest(TDPath));
          /* get successors, in this TD,
                       of the terminated child TD */
          update_frontier_and_LPs(associated_event(event), TD);
      endif;
    else                      /* event belongs to this TD */
      /* get successors of event in this TD */
      update_frontier_and_LPs(event, TD);
      triggeredTD := triggeredTD(event);
      if triggeredTD /= NULL
        /* i.e., event is also Origin event of a child TD */
        then
         /* add child to active children list of this TD */
          activeChildren(TD) :=
                  activeChildren(TD) U {triggeredTD};
          /* update the child TD */
          h_update(associated_event(event), list(triggeredTD));
      endif;
  endif;
  (feasible_set, maxtime) := h_compute_feasible_set(TD);
  return(feasible_set, maxtime);
end h_update.
```

Figure 7: Hierarchical graph update algorithm.

```
PROCEDURE h_compute_feasible_set(TD)
  maxtime(TD):=
     min(min_over_[X in frontier(TD)](sup(X)),
     min_over_[Child in activeChildren(TD)](maxtime(Child)));
  feasible_set := union(frontier(TD),
                        union_over_[Child in activeChildren(TD)]
                          (feasible_set(Child)));
  feasible_set :=
     remove_from(feasible_set, (X | inf(X) > maxtime));
  if (leafTD(TD))
    then
      compute_pairs_longest_paths(feasible_set);
      feasible_set :=
       remove_from(feasible_set,
                   (X | thereExists(X') and LP(X'X) > 0));
  endif;
  feasible_set(TD) := feasible_set;
  return(feasible_set(TD), maxtime(TD));
end h_compute_feasible_set.
```

Figure 8: Hierarchical feasible set computation algorithm.

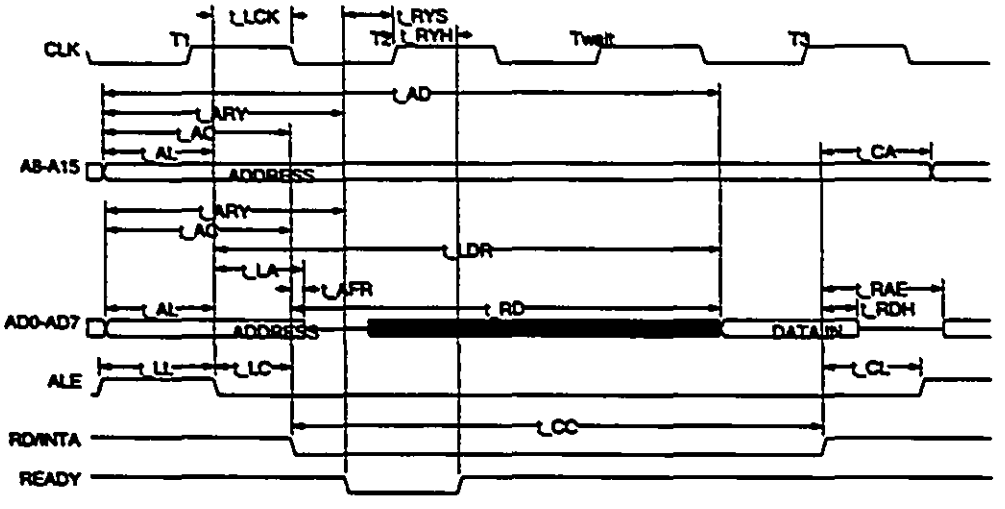


Figure 9: CPU READ - Timing Diagram.

Table 1: CPU READ - Timing Constraints.

Symbol	Parameter	Min.	Max.	Units
T_{CYC}	CLK Cycle Period	320	2000	ns
t_1	CLK Low Time	80		ns
t_2	CLK High Time	120		ns
t_{AL}	Address Valid before Trailing Edge of ALE	110		ns
t_{LA}	Address Hold Time After ALE	100		ns
t_{LL}	ALE Width	140		ns
t_{LCK}	ALE Low During CLK High	100		ns
t_{LC}	Trailing Edge of ALE to Leading Edge of Control	130		ns
t_{AFR}	Address Float After Leading Edge of READ(INTA)		0	ns
t_{AD}	Valid Address to Valid Data In		575	ns
t_{RD}	Read (or INTA) to Valid Data		300	ns
t_{RDH}	Data Hold Time After READ (INTA)	0		ns
t_{RAE}	Trailing Edge of READ to Re-Enabling of Address	150		ns
t_{CA}	Address (A8-A15) Valid After Control	120		ns
t_{DW}	Data Valid to Trailing Edge of WRITE	420		ns
t_{WD}	Data Valid After Trailing Edge of WRITE	100		ns
t_{CC}	Width of Control Low (RD, WR, INTA)	400		ns
t_{CL}	Trailing Edge of Control to Leading Edge of ALE	50		ns
t_{ARY}	READY Valid From Address Valid		220	ns
t_{RYS}	READY Setup Time to Leading Edge of CLK	110		ns
t_{RYH}	READY Hold Time	0		ns
t_{LDR}	ALE to Valid Data In		460	ns
t_{RV}	Control Trailing Edge to Leading of Next Control	400		ns
t_{AC}	Address Valid to Leading Edge of Control	270		ns

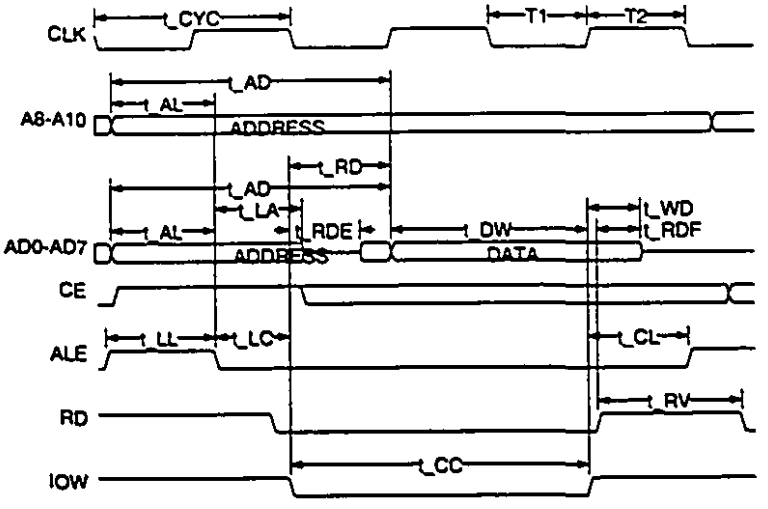


Figure 10: ROM READ - Timing Diagram.

Table 2: ROM READ - Timing Constraints.

Symbol	Parameter	Min.	Max.	Units
T_{CYC}	CLK Cycle Period	320		ns
T_1	CLK Low Time	80		ns
T_2	CLK High Time	120		ns
t_{AL}	Address to Latch Set Up Time	50		ns
t_{LA}	Address Hold Time After Latch	80		ns
t_{LC}	Latch to READ/WRITE Control	100		ns
t_{RD}	Valid Data Out Delay from Read Control		170	ns
t_{AD}	Address Stable to Data Out Valid		400	ns
t_{LL}	Latch Enable Width	100		ns
t_{RDF}	Data Bus Float After Read	0	100	ns
t_{CL}	READ/WRITE Control to Latch Enable	20		ns
t_{CC}	READ/WRITE Control Width	250		ns
t_{DW}	Data In to WRITE Set Up Time	150		ns
t_{WD}	Data In Hold Time After WRITE	10		ns
t_{RV}	Recovery Time between Controls	300		ns
t_{RDE}	Data Out Delay from READ Control	10		ns

Table 3: Event activity for one READ transaction.

Generated by CPU:	CLK	= low	at	0	ns
Generated by CPU:	ALE	= high	at	30	ns
Generated by CPU:	A[8-10]	= 5	at	60	ns
Generated by CPU:	AD[0-7]	= 230	at	60	ns
Generated by CPU:	CE	= low	at	60	ns
Generated by CPU:	CLK	= high	at	140	ns
Generated by CPU:	ALE	= low	at	185	ns
Generated by CPU:	CLK	= low	at	320	ns
Generated by CPU:	CE	= high	at	332	ns
Generated by CPU:	RD	= low	at	332	ns
Generated by CPU:	AD[0-7]	= Z	at	332	ns
Generated by ROM:	AD[0-7]	= U	at	337	ns
Generated by ROM:	AD[0-7]	= 123	at	398	ns
Generated by CPU:	CLK	= high	at	460	ns
Generated by CPU:	CLK	= low	at	640	ns
Generated by CPU:	CLK	= high	at	780	ns
Generated by CPU:	RD	= high	at	845	ns
Generated by ROM:	AD[0-7]	= Z	at	895	ns

Table 4: Run-time performance of DSGRV.

Experiment	Nodes		Edges		time	event/
	CPU	ROM	CPU	ROM	(sec.)	sec.
1 READ	20	20	66	68	5.0	7.2
2 READ flat	38	38	140	142	19.9	3.6
2 READ hgraph	42	42	138	142	10.4	7.0

CHAPTER 3

MODELING AND EXECUTION OF TIMING DIAGRAMS WITH OPTIONAL AND MULTI-MATCH EVENTS

ABSTRACT

We present a tool that captures the interface specification of a hardware module from a set of timing diagrams. The specification is interpreted as an operational model. Upon execution, the model validates the module input events and produces its output events. Our main contribution is the extension of an existing event-based specification method to support the concepts of *optional events* (events that do not always have to match actual event occurrences) and *multi-match events* (events that can match multiple actual event occurrences). These concepts are necessary in specifying the interface behavior of most digital systems.

1 Introduction

An abstraction paradigm that is gaining acceptance in the timing analysis of large systems, is that of an interface specification [1, 2, 3, 4, 5]. This is an event-based description that captures the causality and timing relations between events at the I/O ports of (usually high-level) modules of the system under verification. In [6], the interface specification of a module is captured in the form of a hierarchy of timing diagrams (TDs) [7] and is internally represented by a hierarchical constraint graph. The specification is then interpreted as an executable (simulation) model. During the execution, the constraint graph hierarchy is traversed in order to validate the module input events and produce its output events according to the specifications. The algorithm of [6] is adequate for fully specified values, e.g., 0, 1, or z bit values. In such a model, every specified event *must* be matched once, and only once in a given execution of the TD. However, the specification of certain types of timing constraints (e.g., set-up and hold times) in the context of an event-based model, requires the use of symbolic event values such as *Valid* and *Don't-care*. Events with such values cannot be handled by the approach of [6]. In this paper, we introduce two new event types: *optional events* (events that do not always have to match actual event occurrences) and *multi-match events* (events that can match multiple actual event occurrences), and we consequently extend the execution model of [6]. A concept similar to a multi-match event was proposed in [8] for the synthesis of asynchronous circuits from Signal Transition Graphs.

The rest of the paper is structured as follows: Section 2 presents our model and terminology. Section 3 presents the input validation algorithm for the restricted case of fully specified logic patterns. Sections 4 and 5 address the problems related to optional and multi-match events, respectively. Section 6 extends the system to output events. Section 7 discusses limitations of the system, and Section 8 concludes the presentation.

2 The Model

An interface specification consists of a set of signals and a set of timing constraints. Each signal is composed of an ordered sequence of events, designated as *spec* events. Given a spec event E on a signal S, the notation *next(E)* designates the spec event which is next to E in the sequence of spec events of S.

An event value indicates the value of the corresponding signal after the occurrence of the event. Event values are in the set $V = B \cup \{z, v, u\}$, where B is the domain of the given signal subtype, e.g., $B = \{0, 1\}$ for a bit signal or $B = \{0, \ldots, 255\}$ for an 8-bit bus signal; z stands for high-impedance; v (*Valid*) represents any arbitrary value from B that does not change for a specified period of time, its actual value being irrelevant to the interface specification; and u means *Unspecified*, *Unknown*, or *Don't-care*. It is assumed that for any spec event E, E and next(E) have distinct values. Event values other than u and v are said to be *fully specified values*. A spec event whose value is fully specified is a *fully specified event*. The direction mode of a spec event is "input" or "output". Unidirectional signals have a single mode of spec events (input only or output only). Bidirectional signals can have both modes.

Timing constraints are represented by a directed constraint graph, where nodes represent events, and a directed edge of weight a from node X to node Y represents the timing constraint: $t_Y - t_X \geq a$, with t_X and t_Y representing the occurrence times of events X and Y, respectively. There are two primitives for specifying timing constraints (Fig. 1): *PREC* (precedence) and *CONC* (concurrency). *X PREC(min, max) Y* means that event Y must occur after event X by at least *min* units of time and at most *max* units of time. The *PREC* constraint can be expressed by the two inequalities: $t_Y - t_X \geq min$ and $t_X - t_Y \geq -max$, where *min* and *max* satisfy $0 \leq min \leq max$. The notation *X PREC(min) Y* is used when *max* is not specified (i.e., $max = \infty$). *X CONC(max) Y* means that the occurrence times of X and Y must be separated by at most *max* units of time. This is expressed by the two inequalities: $t_Y - t_X \geq -max$ and $t_X - t_Y \geq -max$, where $max \geq 0$.

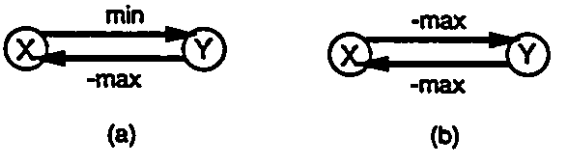


Figure 1: Graph representation. (a) PREC. (b) CONC.

An "Origin" pseudo-event O is created to represent time 0, and a *PREC(0)* relation is added from O to the first spec event of every signal, and between any spec event E and next(E) (if there is a next(E)). Furthermore, an *End* pseudo-event is added with a *PREC(0)* relation from the last event of every signal to the *End* event. The notation *LP(XY)* indicates the weight of the *longest* (i.e., of maximum weight) directed path, from event X to event Y. An event X is said to be a *predecessor* of an

event Y, if X *PREC* Y, or if there exists $Z_1 \ldots Z_n$ such that X *PREC* Z_1, Z_i *PREC* Z_{i+1}, for $i = 1 \ldots n$, and Z_n *PREC* Y. Furthermore, event X is a *generalized predecessor* of Y, if $LP(XY) > 0$. Note that this does not imply that X is a predecessor of Y, e.g., in Fig. 2(a), $LP(XY) = 20$ and X is not a predecessor of Y. The consistency of an interface specification, i.e., whether there exists an assignment of event times such that all constraints are satisfied, is solved [2, 3] by detecting cycles of positive weight in the graph. The satisfiability of safety constraints by causality constraints, i.e., whether all possible time assignments that satisfy the causality constraints also satisfy the safety constraints, can be solved [1, 2] by comparing longest paths between pairs of events in the constraint graph.

Figure 2: Event X is a generalized predecessor of Event Y. (a) Specification. (b) Y occurs at $t = 100$.

We assume an HDL based behavioral simulation environment with concurrent processes, such as in VHDL [9]. Processes relinquish control to a scheduler using "WAIT" instructions that have resume conditions such as timeouts and/or specific event occurrences on signals. The scheduler advances time depending on the temporal latency of the system and gives control back to the processes for which the WAIT conditions have become true. In this context, the executable model of an interface specification is a process which communicates with other processes via its set of I/O signals. Events which occur during the simulation are designated as *actual events*; they must be matched against spec events. The rules for value matching are given in Table 1 for the case of bit values. These rules can be easily generalized to bus signals by substituting $0 \ldots 2^n - 1$, where n is the bus size, for $\{0, 1\}$ in Table 1. When the intent is clear, we will simply use the term *event* to designate a *spec* or an *actual* event.

When an actual event is matched against a spec event X at time t_X, two edges of weight t_X and $-t_X$ from events O to X and X to O, respectively are inserted in the CG. From this, the following result can be easily proven [10].

Lemma 1: The constraint system remains consistent when X is

Table 1: Event value matching.

spec values	matching actual values
0	0
1	1
z	z
v	0, 1, v
u	0, 1, z, v, u

matched at t_X iff t_X is such that: $LP(OX) \leq t_X \leq -LP(XO)$.

In the following, the time interval of Lemma 1 is designated as the *occurrence interval* of event X and is denoted by $[X]$. The notation inf(X) (resp. sup(X)) stands for LP(OX) (resp. -LP(XO)) The *current event* (CE) of a given signal is the spec event following the last occurred spec event on the signal. The CE is nil if all spec events for that signal have occurred. The *current event set* (CES) is the set of current events over the signal set.

3 Validation of Fully Specified Events

In this section, we consider the problem of input validation restricted to *fully specified* events. The validation algorithm is shown in Fig. 3. The process iterates until the current event set is empty. At each iteration, the process WAITs for actual input event activity, and if no such activity occurs before the time T = *max-time*, a timeout occurs. *Max-time* is the smallest sup(X_i), where X_i spans the set of current events. The procedure *update* in Fig. 3 updates the longest paths and the current event set (i.e., if a spec event E of a signal S is matched, CE(S) is assigned the value next(E)).

There are three types of invalid situations. The first type, signaled by the procedure *match-or-error*, is due to the occurrence of an input event that does not match the current spec event of the corresponding signal. Matching is based on signal name, event value and occurrence time, as given by Lemma 1. The second type (*time-out-error*) occurs when the process is woken up due to a timeout (i.e., T = *max-time*) and no input event has occurred. The third type (*precedence-error*) is due to the occurrence of an event Y that violates a generalized precedence relation with respect to some yet unoccurred event X (i.e., X should have occurred

before Y). The effect of Y's occurrence is that sup(X) becomes smaller than the current simulation time. For example, in Fig. 2(b), event Y occurs at $t = 100$, making the sup of the yet unoccurred event X take on the value $sup(X) = 80$ (i.e., smaller than the current time $t = 100$). We say that X is "projected into the past". In order to detect precedence errors, it is sufficient to check whether there exists a current event that has a sup value smaller than the current simulation time. Indeed, since there is a *PREC(0)* relation between any event E and next(E), the sup of the signal's current event is the smallest of the sup of all unoccurred events of the signal.

4 Optional Events

In general, not all spec events have to match an actual event occurrence during execution. An example of this is a data change event E_1 that precedes a clock event E_2 by a required minimum set-up time t_{su} (Fig. 4(a)). However the data change does not need to occur. Such spec events are designated as *optional events*. An event which is not optional is said to be *necessary*. Note that whether a spec event is optional or necessary, cannot always be determined statically. For example if two spec events E and F have values v and 1, respectively, and F is next(E), then F could be optional or necessary, depending on the value of the *actual* event that matches E.

In order to handle optional events, we relax the "timeout error" rule. The new rule states that it is legal to reach a timeout for a spec event E (i.e., to have the current simulation time advance to sup(E) with no actual event activity on the corresponding signal) if the current value of the signal matches the spec event value. It is an error if the values do not match. For example, in Fig. 4(b) assume that E_1 and E_2 are the CE's of the data and clock signals, respectively. Assume further that the context of execution is such that $[E_1] = [10, 90]$ and $[E_2] = [50, 100]$. At time $t = 90$, a timeout occurs for event E_1. The actual signal value of the data must be tentatively matched against the spec value, v, of spec event E_1. If the match is successful (this is the case if the data is one of $\{0, 1, v\}$), the timeout is accepted and the CE for the data signal becomes E_3, else an error is flagged.

The "precedence error" rule needs to be relaxed in a similar manner. The new rule states that it is acceptable for a spec event E to be "projected into the past" (i.e., to have sup(E) smaller than the current simulation

time) if the current value of the signal matches the spec event value. It is an error if the values do not match. Consider again the example of Fig. 4(b) (E_1 and E_2 are the CE's of the data and clock signals, respectively, with $[E_1] = [10, 90]$ and $[E_2] = [50, 100]$). However, assume now that a rising clock event occurs at $t = 70$ (see Fig. 4(c)); this event matches E_2 value-wise and time-wise. After the graph update, $[E_1]$ is equal to $[10, 60]$; E_1 is therefore projected into the past. This is correct as long as the actual signal value of the data matches the value of E_1 (i.e., is one of $\{0, 1, v\}$). Assuming this is the case, the current events of the clock and data signals become E_4 and E_3, respectively. Note that in general, a *sequence* of events on a given signal can be projected into the past (the sequence starts at the signal's CE).

5 Multi-Match Events

A given spec event can match multiple actual events. This is a characteristic of don't care (u) value spec events. For example, in Fig. 4(a), event E_3 expresses the fact that the data signal can undergo a sequence of transitions of arbitrary length (including zero). Spec events such as E_3 are designated as *multi-match events* (note that, by definition, a multi-match event is also an optional event). We interpret constraint edges incident on a multi-match event as being with respect to the first transition (if any) of the matching sequence. Note that by transitivity, the constraint edge of weight t_h in Fig. 4(a) applies to the whole event sequence that eventually matches E_3. However, this would not be the case for constraint edges outgoing from a multi-match event.

Assume that E is a multi-match event (Fig. 5(a)), F is next(E), and the constraints on E (resp. F) are represented, without loss of generality, by two edges of weight a and b (resp. c and d). G represents the rest of the constraint graph. Assume, that E matches a sequence of actual events $E_1 \ldots E_n$. If we knew beforehand the length of this sequence, we could represent the specification as in Fig. 5(b) (note that in order to simplify the notation, we are using the same symbol $E_i, i = 1 \ldots n$ to stand for both the actual event and its associated spec event). Upon matching of events $E_1 \ldots E_n$ at times $t_1 \ldots t_n$ during simulation, the graph of Fig. 5(c) would be obtained. It can be easily shown that, by transitive closure, this graph is equivalent to the one in Fig. 5(d). By "equivalent", we mean that this transformation preserves, at all times, LP(XY), for all X,Y other than the "internal" events $E_2 \ldots E_{n-1}$. Note that the edge of weight $-t_n$ can be dropped because it only affects paths which have the sequence

$E_1 \rightarrow E_n \rightarrow O$ at their tail end; however, this sequence is dominated, in terms of longest paths, by the edge $E_1 O$, of weight $-t_1$ (since $-t_1 > -t_n$). The motivation for dropping the edge of weight $-t_n$ will become clear in the following.

The generalized validation algorithm handles the model in Fig. 5(a) as follows: During the graph initialization that precedes the actual simulation, event E is "split" into E_1 and E_n. In fact E itself stands for E_1, and a new spec event is inserted between E_1 and F to represent E_n. During simulation, when E_1 is the CE, and an actual matching event occurs at time t_1 on the corresponding signal, edges of weight t_1 and $-t_1$ are inserted in the graph, E_1 is considered as occurred, and the CE of the signal is set to E_n. Then, when an actual event matches E_n at time t_i , $t_i > t_1$, a single edge of weight t_i is inserted from O to E_n, and the CE remains equal to E_n. Subsequently, at every successive match of E_n, the weight of this edge is simply increased to the new occurrence time of the matching actual event (and the CE remains equal to E_n). No edge of weight $-t_i$ is inserted because such an edge would reduce $\sup(E_n)$ to t_i, therefore prohibiting any further matches of subsequent actual events against E_n. Finally, E_n is considered as occurred and the CE is updated to $\text{next}(E_n)$, when one of the following occurs: 1) E_n times out, 2) E_n is "projected into the past". These two situations are handled exactly as seen in Section 4.

Note that when E_1 is the CE it can also time out or be projected into the past. The same thing then happens to E_n, if the path which starts at E_1 and which caused the timeout or projection of E_1 into the past passes through the d edge (Fig. 5(d)); if, however, the path goes through the b edge, E_n becomes the CE.

6 Output Event Generation

In this section we extend the system to specifications that contain both input and output events. We need to address two issues: 1) how to choose an output event to be generated at a given point in the simulation, and 2) how to fix the occurrence time of the chosen event. Note that the tool we are presenting in this paper is a utility on which simulation-based timing analysis services can be built. For example, choosing event occurrence times to test "marginal" or "average" conditions could be implemented on top of the system, but its development is outside the scope of this presentation. Therefore, in cases where there are choices to be made, they are made arbitrarily within the timing intervals that maintain consistency

of the constraint system.

Let us first address the problem of choosing an output event: At any point in the simulation, an output event S to be generated must be chosen from the output subset of the current event set. However additional restrictions must be placed on the choice of S: In the simple case of fully specified events, all generalized predecessors of S must have already occurred. If this condition is met, S is said to be a *feasible* output event. However, in the general case, this "feasibility condition" is too strong; for example, in Fig. 4(a), if E_1 were an input event and E_2 an output event, then this condition could possibly prevent the correct generation of E_2, or at best in the case where E_1 or E_2 have a "natural timeout" (such as in Fig. 4(b)), it would result in the generation of E_2 at the latest possible time, which is too restrictive.

The solution to this problem is to define an output event to be feasible if, and only if, all its *necessary* generalized predecessors have occurred. The predicate *feasible-p()* (Fig. 6) precisely formulates the feasibility condition.

Let us now turn to the problem of fixing the occurrence time of a feasible output event E. The occurrence time t_E of E must be chosen from a sub-interval of $[E]$, as given in the following lemma, in which t_c indicates the current simulation time and *smallestSup* is defined as the smallest $\sup(F_i)$, where F_i spans the set of future (unoccurred) necessary events;

Lemma 2: $\max(inf(E), t_c) \leq t_E \leq \min(sup(E), smallestSup)$.

The proof of Lemma 2 stems from the following two facts. 1) Events must occur in forward running time, thus the need for tightening the lower bound of $[E]$ by t_c. 2) t_E must not exceed *smallestSup*, otherwise it would project a *necessary* event into the past. *smallestSup* is computed by *feasible-p()* (Fig. 6).

Note that all the timeout and "projection into the past" rules apply equally well to output events, i.e., it is acceptable for an output event to time out or to be projected into the past, as long as the event's value matches the current value of the signal.

The generalized specification interpreter is given in Fig. 7. The termination criterion of the main loop of the process is the occurrence of the End event (which is treated as a pseudo output event and occurs as soon as it is *feasible-p*; in the case of hierarchical TDs [6], the End event can also be projected into the past due to the occurrence of an event in the following TD). *Out-event* and *out-time* form the output event to be generated in a

given iteration of the process. In the first such iteration, the Origin event (treated as a pseudo output event) is "generated" at time 0. Then, in subsequent iterations, the function *determine-output-event* arbitrarily selects a feasible *out-event* (if any) using the *feasible-p* criterion, and randomly chooses an *out-time* in the interval given by Lemma 2. Then, when the process times out at time *out-time*, the procedure *occur-now-and-update* generates the event with zero delay and updates the graph consequently.

7 Implementation and Results

We interfaced our system [11] to the SHADOW graphic waveform editor developed at Bell-Northern Research Ltd. The editor has a built-in LISP interpreter that allows easy access and modification to the waveforms database. Hierarchical compositions [6] of timing diagrams are specified in a LISP syntax, using two basic primitives: *TDConcat* and *TDConcur*, for sequential and concurrent execution of timing diagrams, respectively.

We have run a number of experiments in which our system simulated VHDL interface models of entities from their hierarchical timing diagram descriptions. The simulation run-time performance of the system averaged 7 processed events per second of CPU time on a SUN 3/260 running Intermetrics VHDL. We have identified timing behaviors that cannot be expressed in the current framework. Consider for example modules which perform a combinational mapping from their level-sensitive latched inputs to their (unlatched) outputs. Then, every time a latched data input changes (i.e., the input multi-match event is matched) *during* the active clock phase, the corresponding output must change accordingly (this change matches the output multi-match event). In order to express this type of behavior, we could define a new type of timing constraint, called a *multi-match follower* (MMF) constraint. We are also in the process of implementing a TDChoice composition primitive, which allows the specification of branching behavior. In order to support this operation, we are extending our algorithm to match multiple TDs in parallel.

8 Conclusion

We have extended the system presented in [6] to specifications containing *optional* and *multi-match* events, i.e., specified events that match at most one, or any number of actual event occurrences, respectively. Our system

captures the interface specification of a hardware module from a set of timing diagrams. The specification is then interpreted as an executable model. We have identified the limitations of the multi-match event model and we are presently working on further generalizations of the type of timing relations that can be expressed. The direct application of our system is in the simulation of specifications. Other applications are in the verification of interfaces using operational models (e.g., [12]).

References

[1] A.R. Martello and S.P. Levitan "Causal timing verification", *First International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, (TAU 90)*, Vancouver, Canada, 1990.

[2] J.A. Brzozowski, T. Gahlinger, and F. Mavaddat, *Consistency and satisfiability of waveform timing specifications*, Research Report CS-88-24, University of Waterloo, 1988.

[3] S.K. Sherman, "Algorithms for timing requirement analysis and generation", *ACM/IEEE Proc. 25th DAC*, pages 724-727, 1988.

[4] A. Kara, R. Rastogi, and K. Kawamura, "TDS: An expert system to automate timing design for interfacing VLSI chips in microcomputer systems", *IEEE Proc. ICCAD-86*, pages 362-365, 1986.

[5] F. Jahanian and A.K.L. Mok, "A graph-theoretic approach for timing analysis and its implementation", *IEEE Transactions on Computers*, C-36(8), August 1987.

[6] K. Khordoc, M. Dufresne, and E. Cerny, "A stimulus/response system based on hierarchical timing diagrams", *IEEE Proc. ICCAD-91*, pages 358-361, 1991.

[7] G. Boriello, *A New Interface Specification Methodology and its Application to Transducer Synthesis*, PhD thesis, University of California, Berkeley, 1988.

[8] C.W. Moon, P.R. Stephan, and R.K. Brayton, "Synthesis of hazard-free asynchronous circuits from graphical specifications", *IEEE Proc. ICCAD-91*, pages 322-325, 1991.

[9] IEEE, *IEEE Standard 1076-1987, VHDL Language Reference Manual*, IEEE, 1987.

[10] K. Khordoc, M. Dufresne, and E. Cerny, "A stimulus/response system based on hierarchical timing diagrams" Publication 770, Dept. I.R.O., Université de Montréal, 1991.

[11] M. Dufresne, K. Khordoc, and E. Cerny, "Using formalized timing diagrams in VHDL simulation", *Proc. Second European Conference on VHDL Methods, pages 24-31, 1991.*

[12] T. Yoneda, K. Nakade, and Y. Tohma, "A fast timing verification method based on the independence of units", *IEEE Proc. 19th FTCS*, pages 134-141, 1989.

```
PROCESS VALIDATE_INPUTS(TD, signals)
  TD_init(TD); /* compute LPs and initialize current events */
  max_time := min_over_[sig in signals]
                        (sup(current_event(sig)))
  while (current_event_set not empty) do
    current_time := get_current_simulation_time();
    activity? := validate_input_events_and_update(signals);
    if ((not activity?) and (T = max_time))
      then time_out_error(signals);
    end if;
    max_time := min_over_[sig in signals]
                          (sup(current_event(sig)))
    loop for sig in signals do
      if (sup(current_event(sig)) < current_time)
        then precedence_error(sig);
      endif;
    end loop;
    timeout := max_time - current_time;
    wait on signals for timeout;
 end while;
end VALIDATE_INPUTS.

PROCEDURE validate_input_events_and_update(signals)
  actual_input_events :=
                query_simulator_for_input_events(signals);
  loop for actual_event in actual_input_events do
    spec_event := current_event(event_signal(actual_event));
    match_or_error(spec_event, actual_event, current_time);
    update(spec_event, current_time);
  end loop;
  return(actual_input_events);
end validate_input_events_and_update.
```

Figure 3: Restricted input validation algorithm.

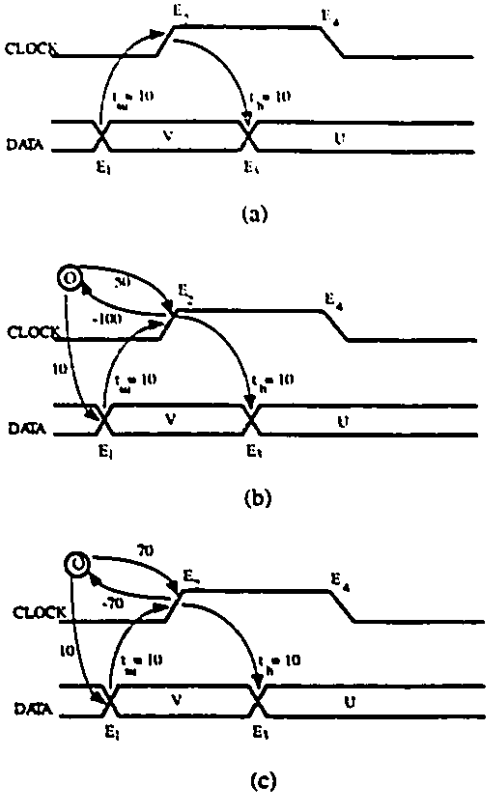


Figure 4: A timing diagram with u and v valued events. (a) Specification. (b) Timeout for E_1 at $t = 90$. (c) Rising clock at $t = 70$.

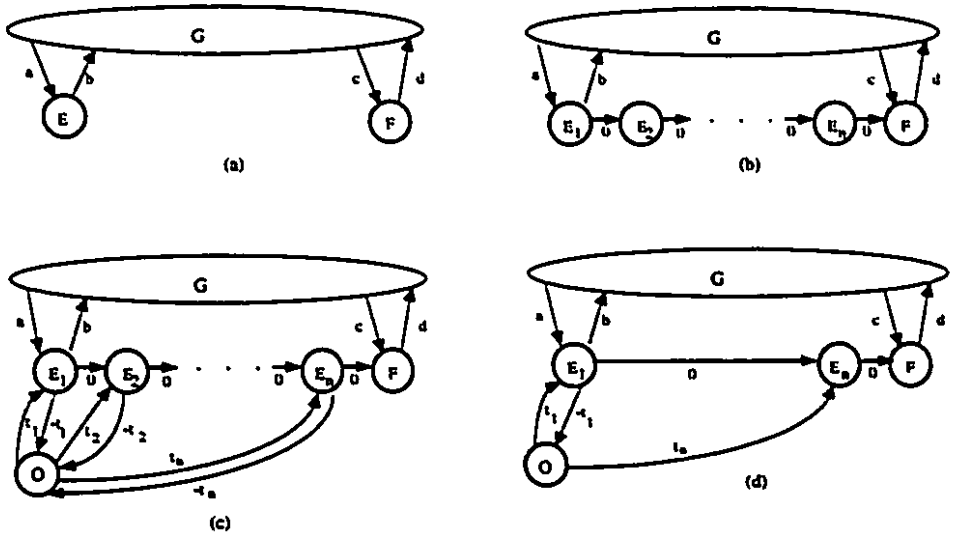


Figure 5: The multi-match event model. (a) Specification. (b) Equivalent representation. (c) Actual events occurrences. (d) Transitive closure.

```
function feasible_p(E)
  feasible := true;
  smallestSup := +infinity;
  loop for sig in (signals - {sig(E)}) do
    loop for F from CE(sig) then next(F)
         and while (inf(F) <= sup(E)) /* efficient exit test*/
      do
        if (not(match(F, signal_value(sig)))) /* F necessary*/
          then LP(FE) := compute_LP(F,E)
               if (LP(FE) > 0)
                 then feasible := false; /*do not generate E */
                 else  smallestSup := min(sup(F), smallestSup);
               end if;
               exit;      /* exit inner loop */
        end if;
    end loop;
    if (feasible = false) then exit;    /* exit outer loop */
  end loop;
  return(feasible, smallestSup);
end feasible_p.
```

Figure 6: Feasible output event predicate.

```
PROCESS EXECUTE_SPECIFICATION(TD, signals)
  TD_init(TD); out_event := origin(TD);
  out_time := 0; max_time := +infinity;
  while (End_event(TD) not occurred) do
    current_time := get_current_simulation_time();
    if (((T = out_time) and (out_event /= NIL)))
        /* make X occur with zero delay */
      then occur_now_and_update(out_event);
    end if;
    validate_input_events_and_update(signals);
    max_time := min_over_[sig in signals]
                    (sup(current_event(sig)))
    validate_timed_out_and_projected_events(signals);
    (out_event, out_time) := determine_output_event(signals)
    timeout := max_time - current_time;
    wait on signals for timeout;
 end while;
end EXECUTE_SPECIFICATION.

PROCEDURE validate_timed_out_and_projected_events(signals)
  loop for sig in signals do
    loop for event from current_event(sig) then next(event) do
        if  (sup(event) <= current_time)
          then match_values_or_error(event, sig);
          else set_current_event(sig, event);
               exit;  /* exit from inner loop */
        end if;
    end loop;
  end loop;
end validate_timed_out_and_projected_events.
```

Figure 7: The Generalized Specification Interpreter.

CHAPTER 4

INTEGRATING BEHAVIOR AND TIMING IN EXECUTABLE SPECIFICATIONS

ABSTRACT

We present a modeling methodology and tool set for the rapid development of executable HDL models. The method is based on the separate capture of interface specifications, functional specifications and the relation between them. HDL models are generated in a *layered* fashion, at different levels of abstraction, in which layers can be easily inserted and removed, thus facilitating the validation of different aspects of the design. *HDL interface models* are automatically generated from the specifications.

1 Introduction

Informal specifications are often unclear, ambiguous and incomplete. Executable HDL models are useful in formalizing, experimenting with, and "animating" specifications; such models can become an integral part of the documentation generated at product inception and act as a golden reference for understanding the specifications. Then, as the system is designed, it must be validated against the specifications. The executable HDL model thus continues to act as a golden reference throughout the design cycle. Formal verification methods, are useful in that they provide a complete "coverage" with respect to the model being verified. However, these techniques are limited to partial models of small size relative to the total state space of the design. They are usually complemented by simulation techniques, e.g., the implementation is simulated against the specification and the results are compared. Furthermore, the designed system must be simulated in its environment in order to verify whether the behavior is as expected (i.e., *integration testing*). In order to achieve this, executable models of standard, but often quite complex, off-the-shelf VLSI components must be developed easily and rapidly.

Integration testing does not proceed in a monolithic fashion (simulation would be too time consuming, huge amounts of useless information would have to be browsed through, etc.). In practice, different aspects of the system need to be verified, e.g., the functional behavior, the interface and timing behavior, leading to the need for different models. This often results in the ad-hoc development of a multitude of models, unrelated to each other, implying inconsistencies between the different views of the system, and making the final phases of integration testing impossible (since these different models cannot be "glued" together).

Of particular importance to integration testing is *interface verification*. Interface specifications capture the fact that components are accessed in specific ways, e.g., in operational units called "interface operations", or "bus cycles", such as FETCH, READ, WRITE cycles etc. Each interface operation consists of specific event sequences related by timing constraints. Interface analysis methods [1, 2, 3, 4, 5] address the problem of verifying that two interface specifications are compatible with each other. However, in order to use these techniques, the interface specification of the designed system must be extracted from the implementation. This is a non-trivial problem for which no standard automatic procedure is known (except for the special case of strictly synchronous interfaces), and it is usually solved by manual techniques that introduce unverified assumptions; this in turn

affects the degree of confidence in the result.

Although the above type of interface analysis is valuable, it must be complemented by other techniques, e.g., running simulations of the system's implementation against *HDL interface models* of the system's environment. These *interface models* are behavioral HDL programs derived from the interface specifications of the components that form the system's environment. The interface model of a component consists in "on-the-fly parsing" of events received at the component's I/O ports, sequencing the model through its state transitions based on the result of this parsing, detecting incorrect, or ill-formed interface operations (bus cycles), verifying that all timing constraints at the input of the component are met, and driving the component outputs with appropriate delays. In the rest of this paper, the term *interface model* will englobe both of the checking and driving aspects. Unfortunately, developing HDL interface models is a tedious, time-consuming and error-prone tasks. The developed code must usually make heavy use of process synchronization primitives (e.g. WAIT statements) and is hard to debug. It is also very difficult to ensure that the model is complete, e.g., whether all constraints are checked under all relevant event sequences.

In this paper we present the modeling methodology and tool set that we have developed in response to the above problems:

- Our approach allows the rapid development of executable HDL models.
- The method is based on the separate capture of interface specifications, functional specifications and the relations between these two forms.
- The methodology and tool set allow the generation of HDL models in a *layered* fashion, at different levels of abstraction, in which layers can be easily "plugged in" or removed, thus facilitating the validation of different aspects of the design.
- *HDL interface models* are automatically generated from the above specifications.

There are three major components in the tool set.

1. The specification capture tools: the hierarchical *timing diagram editor* graphically captures interface specifications. The hierarchical

EFSM editor[1] (Extended Finite State Machine) captures functional specifications. The *functional link editor* captures (in a mix of graphics and text) the relationships between the functional and interface aspects of the specifications.

2. The *model generator* produces executable HDL models (more specifically VHDL [7] models), at the desired level of abstraction, from the captured specifications.

3. The run-time tools: the *timing diagram interpreter* (TDI) and *EFSM interpreter* implement, during the simulation run-time, the executable semantics of the captured specifications.

The rest of this paper is structured as follows: Section 2 presents the interface specification method and illustrates it on an example. Section 3 explains the algorithm that controls the execution of the timing diagram hierarchy. Section 4 presents a functional specification method for entities with simple internal control-flow; the method is illustrated on an example. Section 5 extends the modeling approach to arbitrary behaviors and illustrates it on an example. Section 6 reviews related work and puts our contribution in that perspective. Finally, Section 7 concludes the presentation by discussing some future orientations of our work.

2 Interface Specifications

2.1 Timing Diagrams

A timing diagram (TD) specification consists of a set of signals and a set of timing constraints between signal transitions. Each signal consists of an ordered sequence of events, designated as *spec* events. The direction mode of a spec event is "input" or "output". Unidirectional signals have a single mode of spec events (input only or output only). Bidirectional signals can have events of both modes. Event values are in the set $V = B \cup \{z, v, u\}$, where B is the domain of the given signal subtype, e.g., $B = \{0, 1\}$ for a bit signal or $B = \{0, \ldots, 255\}$ for an 8-bit bus signal; z stands for high-impedance; v (*Valid*) represents any arbitrary value from B that does not change for a specified period of time, its actual value being irrelevant to the interface specification; and u means *Unspecified*, *Unknown*, or *Don't-care*.

[1] A graphic interface is planned. At the present time, the functional specifications are captured in textual format only.

Timing constraints capture min/max time relationships between events, and are represented by a directed constraint graph in which nodes represent events, and a directed edge of weight a from node X to node Y represents the timing constraint: $t_Y - t_X \geq a$, where t_X and t_Y are the occurrence times of events X and Y, respectively.

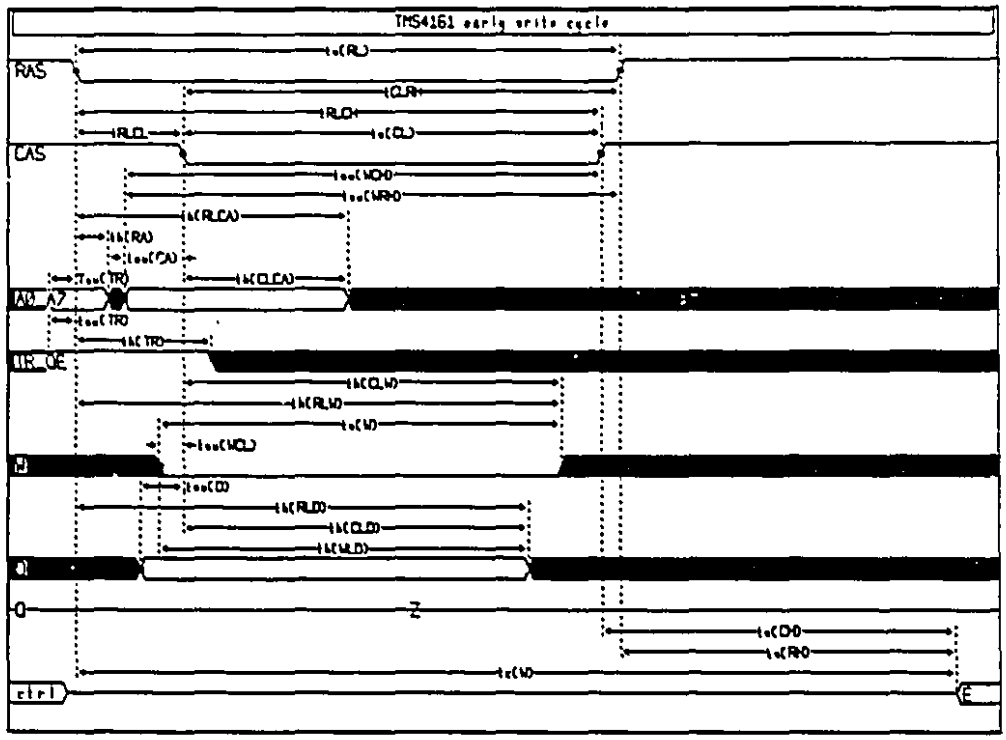


Figure 1: VRAM Write cycle.

The Timing diagram specification is captured graphically using the *SHADOW* waveform editor (e.g., Fig. 1, Write cycle of a dynamic memory [11]) developed at Bell-Northern Research Ltd. The specification is compiled into an executable model. During execution (i.e., simulation), the model validates its input events and produces its output events according to the logic and timing specifications of the timing diagram. Events which occur during the simulation are designated as *actual events*. An actual event is a triplet *(signal, value, time)*, i.e., the signal on which the event occurred, the new value of the signal, and the time of occurrence of the event, respectively. *Match errors* are flagged when actual events cannot be matched (in terms of timing, or logic value) to the specifications; this is the "checker" aspect of the executable model.

A spec event of value v is said to be an *optional event* [9]; such an event does not necessarily have to match an actual event occurrence. A

spec event of value u is said to be a *multi-match event* [9]; such an event can match a sequence of actual transitions of arbitrary length (including zero). For example, in Fig. 2(a), E_1 is an optional event, and E_3 is a multi-match event.

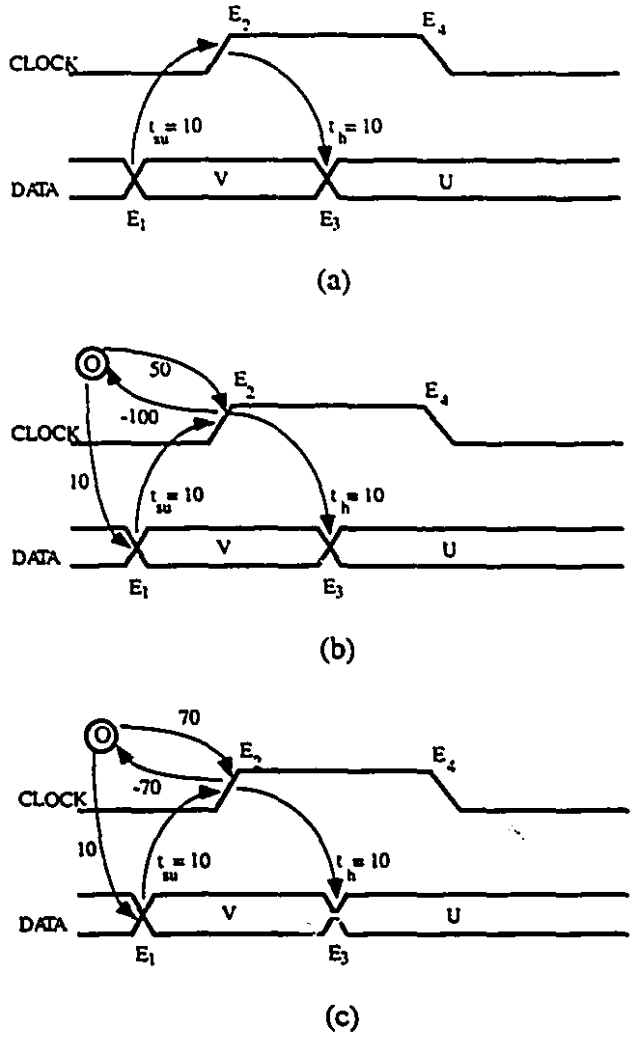


Figure 2: A timing diagram with u and v valued events. (a) Specification. (b) Execution context before clock event. (c) Rising clock at $t = 70$.

Simpler alternatives to optional and multi-match events were considered, such as, for example, defining *stability windows* for "data" signals and restricting the *spec event* concept to "clock" and "control" signals. However, the present model was chosen for the following reasons:

- It is not always possible to easily make the difference between clock,

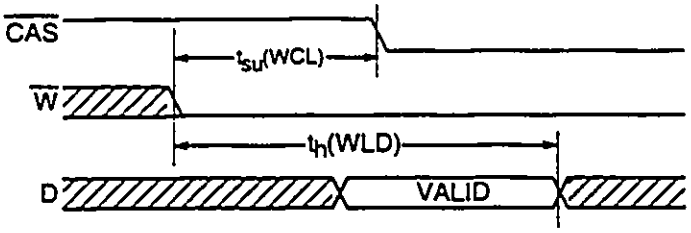


Figure 3: Excerpt from VRAM Write cycle.

control and data.

- The model with optional and multi-match events expresses more general constraints.

  The two above points are illustrated in Fig. 3 (extracted from the Write cycle of Fig. 1). At the start of the cycle, the $\overline{W}$ signal has a "don't care" value and it has a setup time $t_{su}(WCL)$ with respect to the falling edge of $\overline{CAS}$ (i.e. , $\overline{W}$ acts as a "data" signal). However the D line (input data to the memory) has a hold time $t_h(WLD)$ with respect to the falling edge of $\overline{W}$ (i.e., $\overline{W}$ acts as a "clock" signal)

- Due to the consistent event based semantics, our model offers a unified framework for linking procedures to both "clock" and "data" events (this will be explained in Section 4).

The main data structure supporting the execution is a timing constraint graph (also called event graph) [6, 8], extended as in [9], for the processing of optional and multi-match events. Details relative to the propagation of timing constraints in the event graph, using longest path computations, can be found in [10]. The rules for matching the value of an actual event to that of a spec event, are given in Table 1 for the case of bit values. These rules can be easily generalized to bus signals by substituting $0 \ldots 2^n - 1$, where n is the bus size, for $\{0, 1\}$ in Table 1. When the intent is clear, we will simply use the term *event* to designate a *spec* or an *actual* event.

2.2 Composing Timing Diagrams

Timing diagrams can be composed recursively to describe interface specifications. The composition operators are: *Concatenation*, *Loop*, *Concurrency*, and *Choice*. In Fig. 4, symbols $A_1, \ldots, A_n$ refer to timing diagrams

Table 1: Event value matching.

spec values	matching actual values
0	0
1	1
z	z
v	0, 1, v
u	0, 1, z, v, u

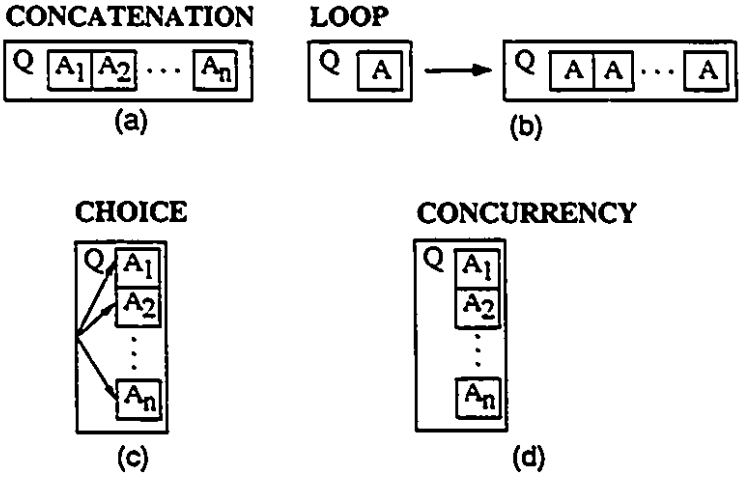


Figure 4: Timing diagram composition operations.

that are composed to form a more complex timing diagram, Q; naturally, $A_1, \ldots, A_n$ can themselves be the result of other compositions, etc.

For all operators, except *Choice*, a *match error* in one of $A_1, \ldots, A_n$, unconditionally translates into an error in Q. In the case of *Choice*, the semantics is slightly more complex (explained below). The interpretative semantics of each composition operation is described in the following.

Concatenation: $A_1, \ldots, A_n$ are defined on the same set of signals. A_1 starts when Q starts. A_{i+1} starts when A_i terminates. Q terminates when A_n terminates.

Loop: The semantics are similar to *Concatenation* with $A_1, \ldots, A_n$ being identical copies. Two modes are supported: *Loop* with a fixed number of iterations, and infinite *Loop*.

Choice: $A_1, \ldots, A_n$ are defined on the same set of signals; they represent alternative (branching) behaviors. $A_1, \ldots, A_n$ start when Q starts.

Whenever a match error is found in an A_i, A_i is terminated. If one of $A_1, \ldots, A_n$ terminates *free* of match errors, then Q immediately terminates free of match errors, else Q terminates with a match error.

Concurrency: $A_1, \ldots, A_n$ are defined on mutually disjoint subsets of signals; they represent concurrent activity taking place on these subsets. $A_1, \ldots, A_n$ start when Q starts. Q terminates when all of $A_1, \ldots, A_n$ terminate.

2.3 Example

The *control-flow* of an interface specification describes what timing diagram is to be "executed" next, what events can be generated/received and at what time. The composition operators presented above allow the description of a subclass of interface behaviors for which the outcomes of the high-level control-flow branches (namely what TD to execute in a Choice operation) are determined by the *environment* of the entity. Dynamic RAMs are good candidates for modeling with this approach because, in addition to meeting this interface control-flow criterion, these devices have a quite complex interface behavior, which however, can be easily expressed using hierarchical timing diagrams.

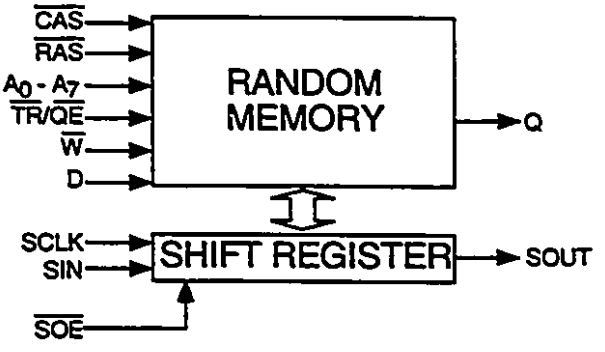


Figure 5: VRAM block diagram.

As an example, we show the modeling of the interface behavior of the TMS4161 [11] dual-port Video RAM (VRAM). A block diagram of the device is given in Fig. 5. The random access port behaves as in a normal dynamic random access memory (it supports access cycles such as READ, WRITE, READ-MODIFY-WRITE etc.). It is controlled by $\overline{RAS}$ (Row Address Strobe), $\overline{CAS}$ (Column Address Strobe) and $\overline{W}$ (Write). D, $A_0 - A_7$ and Q are the input Data, Address and output data buses, respectively; Q can be tri-stated under the control of $\overline{QE}$ (Q Enable). The

sequential access port behaves as a shift register controlled by *SCLK* (Shift Clock). *SIN* and *SOUT* are the register Shift In and Shift Out data lines, respectively. *SOUT* can be tri-stated under the control of $\overline{SOE}$. A *transfer cycle* allows to internally "parallel load" the shift register of the sequential access port with a given row of the random access memory. A transfer cycle is determined by a low $\overline{TR}$ (i.e., a low $\overline{TR/QE}$ during the $\overline{RAS}$ falling edge; $\overline{TR}$ and $\overline{QE}$ are multiplexed into a single line that is interpreted as $\overline{TR}$ at the falling edge of $\overline{RAS}$, and as $\overline{QE}$ the rest of the time). The two ports operate concurrently and asynchronously to each other, except during transfer cycles. During such a cycle, there are timing constraints between control signals of the two ports (more specifically between $\overline{RAS}$ and *SCLK*) and the behavior of*SOUT* is different from the case without transfer (as a new row of data is loaded into the shift register).

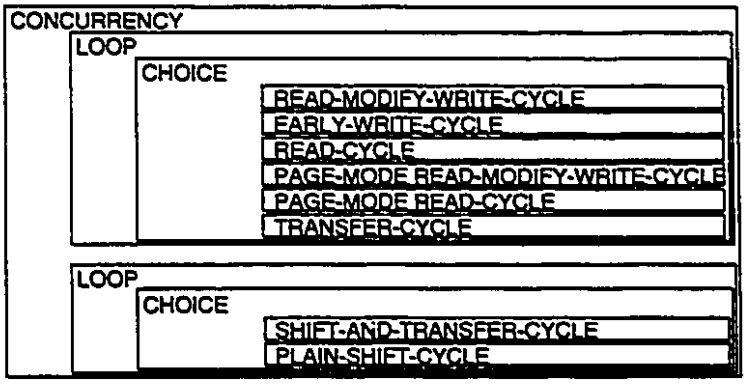


Figure 6: VRAM interface specification.

A high-level view of the VRAM interface, modeled using the TD composition operators, is shown in Fig. 6. The model top-level puts the random port in *Concurrency* with the sequential port. The random port is modeled as an infinite *Loop* of a *Choice* of random access cycles (each of these cycles is described in [11] by a timing diagram). The READ, WRITE and TRANSFER cycles are modeled as leaf-level timing diagrams (e.g., WRITE is shown in Fig. 1). Each "page mode" cycle shown in Fig. 6 (i.e., PAGE-MODE-READ and PAGE-MODE-READ-MODIFY-WRITE) is in fact broken down into three subcycles (beginning, middle, and end of the given page mode cycle). Each of these subcycles is put in as a direct child of the random access *Choice* TD, in place of the original page mode cycle.

The sequential port is implemented as a Choice between a "normal shift" cycle (PLAIN-SHIFT) and a "shift cycle during transfer" (SHIFT-AND-TRANSFER). These two cycles discriminate on the value of $\overline{TR/QE}$ on the falling edge of $\overline{RAS}$ to determine whether there is a transfer. (To

work around the rule of disjoint signal subsets in the *Concurrency* operator, we created a wrapper around the VRAM entity which forked an extra copy of each of the $\overline{RAS}$ and $\overline{TR/QE}$ signals.)

In the next section, we present the algorithm that controls the execution of the timing diagram hierarchy. Then, in Section 4 and Section 5 we extend the modeling approach to include the functional behavior of the modeled entity.

3 The Timing Diagram Interpreter

3.1 Basic Concepts

During simulation, the Timing Diagram Interpreter (TDI) validates the input events of the modeled entity, and generates its output events, according to the (hierarchical) timing diagram specifications. In order to explain the TDI algorithm, a few definitions are useful.

The *current event* of a signal is the spec event following the last occurred spec event on the signal. The *current event set* is the set of current events over the signal set.

The notation *LP(XY)* denotes the weight of the *longest* (i.e., maximum weight) directed path in the timing constraint graph (associated with some leaf TD) from event node X to event node Y. A valid time interval of occurrence, denoted $[E]$, is associated with each spec event E. The lower and upper bounds of this interval are denoted *inf(E)* and *sup(E)*, respectively; they are computed as LP(OE) and $-$LP(EO), respectively, where O is an "Origin" pseudo-event representing the start time of the TD [10].

When an event E occurs on a signal S at the current simulation, it can make the *sup* time of the yet unoccurred spec event sequence $P_1 \ldots P_n$, on a signal S' other than S, become smaller than the current time, i.e., $sup(P_i) <$ *current time*, for $i = 1 \ldots n$. We say that $P_1 \ldots P_n$ are "projected into the past". Each such spec event $P_i, i = 1 \ldots n$ is said to be *legally projectable* (or simply *"projectable"*), if, and only if, its value matches the current actual value of S', and $P_{(i-1)}.i > 1$ is projectable. Consider, for example, Fig. 2(b) in which E_1 and E_2 are the current events of the data and clock signals, respectively, with $[E_1] = [10, 90]$ and $[E_2] = [50, 100]$. Assume that a rising clock event occurs at $t = 70$ (Fig. 2(c)); this event matches E_2 value-wise and time-wise. After the

graph update, $[E_1]$ is equal to $[10, 60]$; E_1 is therefore projected into the past. This is correct as long as the actual signal value of the data matches the value of E_1 (i.e., is one of $\{0, 1, v\}$). Assuming this is the case, the current events of the clock and data signals become E_4 and E_3, respectively. Note that in general, a *sequence* of events on a given signal can be projected into the past (the sequence starts at the current event of the signal).

The definition of a projectable event is extended to a timing diagram: a TD is projectable if all its unoccurred spec events are projectable.

A *current event set*, *match error* and *projectable* attribute are associated with each TD in the hierarchy. The task of the TDI consists essentially in updating these attributes when events occur. This update is done in a single (post-order) traversal of the TD hierarchy for each actual event occurrence. An *update method* is associated with each TD class; the classes are: Leaf, Choice, Concurrency, Concatenation, and Loop. The input parameters of the update methods are: *self* (the TD object to update) and *event* (the actual event being validated). In the following, we explain the update method of each TD class.

3.2 Leaf Update

The leaf update method is shown in Fig. 7. The *match error* flag is true in any one of the following cases:

1. The actual event does not match the spec event (i.e., either their values do not match, or the occurrence time of the actual event is not within the interval of occurrence of the corresponding spec event).
2. The actual event projects into the past a non-projectable spec event.

The rules for *current events* are as follows:

1. The *current event* of each signal is initially set to the first spec event in the spec event sequence of that signal in the leaf.
2. When an input multi-match event is *current*, it remains so until it is projected into the past.
3. When a current event (other than an input multi-match event) matches an actual event during the execution, the next spec event in the spec

event sequence of the signal (or NIL if there is no next event) becomes the current event.

4. For all signals which have events that are projected into the past (due to the occurrence of an event on another signal), their spec event is advanced to the first event of the signal that has a *sup* time bound greater than, or equal to, the current time.

The procedure updates the longest paths in the leaf event graph, as explained in [10].

3.3 Hierarchical Update

In this section, we explain the update methods for each non-leaf TD class. In the following, the predicate *empty(TD)* returns True for a *TD* that has an empty current event set.

The update method for a *Choice* TD is given in Fig. 8. The TD maintains a list of *active children*. A child is *active* if it has no match errors and is not empty. The *Choice* TD performs a recursive *update* on all its active children (i.e., all still matching branches must be validated). A child is de-activated (i.e., removed from the active children list) when its match error attribute becomes true (as a result of the child update). The *Choice* TD is *projectable* if one of its active children is projectable. The TD sets its *match error* attribute to True if all its active children have match errors. The *current event set* of the TD is obtained by appending together the current event sets of its active children. The current event set is emptied (set to NIL), signifying successful termination, when a child of the *Choice* TD becomes empty and has no match errors.

The update method for a *Concurrency* TD is given in Fig. 9. The children of a *Concurrency* TD are defined over disjoint signal subsets. Therefore, the recursive *update* is performed for the only child defined over the concerned signal. The *match error* attribute of the *Concurrency* TD is set to True if the child is empty before the update, or if the child declares a match error as a result of the update. The *Concurrency* TD is *projectable* if all its children are projectable. The *current event set* of the TD is obtained by appending together the current event sets of its children.

The update method for a *Concatenation* TD is given in Fig. 10. The TD maintains a pointer to its *current child* (i.e., the child which is presently executing). The *current event set* of the *Concatenation* TD is nominally equal to that of its current child. In addition, if the current child

is *projectable*, the current event set of the *Concatenation* TD is extended (*extend-cur-events* in Fig. 10) to the *next child* of the *Concatenation* TD. This extension[2] is done only for signals which have exhausted their spec events in the current child.

Actual events are matched in the current child until it becomes *empty*, or in the case of a projectable current child, until an actual event occurs on a signal which has a spec event in the next child extension of the current event set (in this latter case, the current child is projected into the past). The current child pointer is then updated to the next child. The *match error* attribute of the *Concatenation* TD is set to that of the child in which the event was matched. The *Concatenation* TD is *projectable* if its current child is its *last* child and is projectable. Finally, note that by enforcing the reasonable assumption that each leaf timing diagram contains at least one "necessary" (i.e., neither optional nor multi-match) event, the current event set extension discussed above need not go beyond the *next* child of the *Concatenation* TD.

The *Loop* class is a subclass of *Concatenation*. The *Loop* update method is exactly the same as that in Fig. 10. The differences in the processing of a *Loop* TD with respect to a *Concatenation* are as follows:

1. The children list is implemented as a circular list of two identical child subtrees[3].

2. The *SET* method for advancing the *current child* (Fig. 10) swaps the current and next pointers and performs an appropriate re-initialization of the former *current* sub-tree, so that it can be re-used. In the case of the fixed number of iterations subclass of *Loop*, the *SET* method also increments an iteration counter.

3. The *next-child* accessor and the *is-last-child* predicate (Fig. 10) are specialized methods for the *Loop* class. In the case of the fixed number of iterations subclass of *Loop*, they test the iteration count. In the case of the infinite *Loop* subclass, they return the successor child in the circular list and *False*, respectively.

[2]To be precise, the extension is actually done for signals which have exhausted their spec events in at least one *Choice* branch in the subtree rooted at the current child of the *Concatenation* TD. In the actual implementation, the TDs propagate during the update traversal, signal attributes which indicate this information; these details are omitted from the update methods of Figs 7 to 10.

[3]Two instances, one for the *current* child, and one for the *next* child, are sufficient as a result of the assumption made above, of one necessary event per leaf timing diagram.

Finally, note that a hierarchical event trace is optionally maintained by the TDI. These details are omitted from the pseudo-code of Figs 7 to 10. To maintain this trace, a *history instance* of the hierarchy is progressively built as TDs (at any level of the hierarchy) are matched. This hierarchical history instance differs from the original hierarchical specification in that events have fixed time-value pairs (multi-match events have a list of time-value pairs in general), loops are "unfolded" (i.e., their subtrees are instantiated as many times as necessary), and choices are "linearized" (only the matching branch is kept). The user has control over the "trace period" by specifying, for each Loop TD, the number of iterations for which the trace is to be kept before it is overwritten by subsequent iterations. The user also has control over the "trace density" by specifying which TDs in the hierarchy ought to be considered leafs from a trace history point of view.

3.4 The Top-Level Process

The top-level control loop of the TDI algorithm is shown in Fig. 11. The *TD* parameter is the root of the timing diagram hierarchy. The process iterates until the current event set is empty, or there is a match error. At each iteration, the following is performed:

1. The process queries the simulator for actual input events and collects the result in the list *actual-events*.

2. If an output event must be generated at the current time (this was determined in some previous iteration of the process), the event is made to occur with zero delay (*occur-now*). The event is also appended to the *actual-events* list.

3. For each signal that has an input spec event E in *cur-events(TD)* (i.e., the current event set) with sup(E) equal to the current time, and that has no event in the *actual-events* list, the procedure *append-time-out-events* appends a "fake" actual event (with value equal to the current value of the signal and time equal to the current time) to the *actual-events* list. This is done to force an *update* for this signal (and as a possible result flag errors, e.g., for events that should have, but have not actually occurred).

4. A recursive *update* (i.e., Figs 7 to 10) is performed for each event in *actual-events*.

5. The procedure *compute-output-event* chooses an output event for generation. This is done by randomly assigning an occurrence time (*time(E)*) in the time interval $[E]$, for each output event E in *cur-events*, and then selecting the output event with the smallest assigned occurrence time.

6. The process WAITs for actual input event activity, or for a time-out to occur, where the time-out is computed as the minimum of the output event time and the smallest *sup* in the current event set.

4 Procedural linking

In this section, we extend the modeling approach to include the internal (i.e., functional) behavior of the modeled entity. We concentrate on the class of entities which are characterized by the following two properties: 1- the interface control-flow is dominated by the environment, and 2- the internal control-flow follows quite closely the interface control-flow. For these entities, what remains to be described in order to obtain a complete model, can be achieved with the help of the simple, yet powerful paradigm of linking procedures and functions to events (this will be generically referred to as "procedural linking" in the rest of this document). This linking is specified by pointing to the desired "trigger event" in the timing diagram editor, and by specifying the name and interface of the procedure or function to be linked (the body is edited separately using a text editor).

We distinguish two classes of procedural linking, defined in the following.

1. Procedures (linked to input or output events): A *procedure* linked to an input (resp. output) event (referred to as the "trigger event") is called by the Timing Diagram Interpreter when it matches (resp. generates) the event during the simulation. The procedure is called for its side effects. The parameters of the procedure can be signal names (they stand for the signal values at the time the procedure is called) and/or variables of the internal model. The procedure is allowed to modify only these variables. Such procedures are often used to provide operands to the data-flow operations of the internal behavior.

   For example, in Fig. 12, the *read_column* procedure, which is linked to the $\overline{CAS}$ falling edge event, stores the value of the column address into the *column* variable. This variable will then be used as

an operand to the memory access operation (which is essentially a data-flow operation). More generally, linked procedures can modify variables that determine the control-flow of the internal model. Note, however, that this allows only simple internal control-flow (i.e., that differs only slightly from the interface control-flow).

2. Output computation functions (linked to output events only): A *function* linked to an output trigger event, is called by the Timing Diagram Interpreter when this latter generates the event during simulation. The function returns a signal value to the TDI; the TDI uses this value to generate the event. Such functions essentially model the data-flow operations of the internal behavior (e.g., some arithmetic computation, or memory/register access).

   For example, in Fig. 12, the function *compute_data*, which is linked to the *Q Valid* event, is called by the TDI when the *Q* data must be put on the bus. The function performs the memory access operation and returns the value to the TDI.

   More generally, there can be some control-flow in the function, e.g., branching to different computations depending on the value of some state variable (again, this is typically suitable for entities with an internal control-flow that diverges only slightly from the interface control-flow).

In the case of a trigger event that is matched by the TDI in multiple branches of a Choice operation, the attached procedure or function is executed only once. When events are projected into the past, the procedures or functions attached to them are not executed. The linked procedures and functions do not manage time, nor process synchronization, time outs etc. (e.g., they do not use WAIT statements). These aspects are handled by the TDI; this facilitates the quick development of executable models.

Memory devices are typical examples of entities with an internal control-flow that follows closely the interface control-flow. As a result, it is quite simple to obtain a complete behavioral model of the VRAM by augmenting its interface model (given in Section 2.3) with appropriate procedural linking. For example, we conducted a case study wherein we assigned two students the task of developing a complete VHDL behavioral model for the VRAM, including timing checks, using the technical specifications of [11]. The first student, who had more than a year of experience in VHDL behavioral model development was asked to develop the VRAM model using the VHDL language only. The second student, who had no prior knowledge of VHDL, nor of the timing diagram tools, was asked to

use the hierarchical timing diagram editor, procedural linker and model generator to develop the behavioral model. Apart from the difference in VHDL experience, the two students had similar backgrounds. At the end of the semester, the first student had written about 1,000 lines of VHDL code; this code modeled only the "simple" cycles (i.e., it excluded the page mode cycles.) The second student had specified the VRAM interface, including the page mode cycles, as described in Section 2.3, and had written less than 35 lines of VHDL code in order to complete the behavioral model. This code was essentially made up of small, easy to debug procedures and functions (e.g., Fig. 12).

5 A Complete Approach to Modeling

The requirements for extending the modeling paradigm to arbitrary behaviors, are as follows:

(R1) Allow the control-flow of the interface behavior to be governed by the internal behavior (without, on the other hand, losing the capability of letting the environment govern the control-flow, if desired, as was done in Section 2.2).

(R2) Offer full-fledged modeling capabilities for the internal behavior (both control-flow and data-flow) using an easy and intuitive paradigm for the capture of specifications.

(R3) Define a clear and simple model for the interelation and synchronization between the internal behavior and the interface behavior.

We are presently conducting a modeling case study on the 8085 processor [13], using the following solutions to the above requirements.

(S1) To achieve Requirement R1, a *choose function* and a *loop predicate* are linked to the CHOICE and LOOP timing diagram composition operators, respectively. The input parameters of these functions and predicates can be any subset of the state variables of the internal model. The *choose function* returns the instance name of the child of the Choice TD to be executed by the TDI. The boolean value returned by the *loop predicate* indicates whether a new iteration is to be executed by the TDI. In the case of a Choice TD with no *choose function*, the semantics are as in Sections 2 to 4, i.e., parallel matching of the child TDs.

(S2) The modeling of the internal behavior (Requirement R2) is done using an Extended Finite State Machine (EFSM) model. State transitions are labeled with conditions on EFSM variables. Each state of the EFSM contains a list of actions to be performed "in parallel". These actions consist of variable (register) assignments and simple built-in operations (such as Add, Shift, etc.). Note that this level of abstraction is higher than RTL (Register Transfer Level), in that the states of the EFSM can have variable time durations. For example, in the case of a synchronous entity, different states can require different numbers of clock cycles to execute.

(S3) An EFSM state can be labeled with a *synchronization point*; this indicates that the TDI must take control of the model execution once the EFSM actions in this state are performed. Synchronization points can also label TDs of the timing diagram hierarchy. When a timing diagram labeled with a synchronization point terminates its execution, it must return control to the EFSM.

Operationally, the cooperative execution of the EFSM (which models the internal behavior) and the TDI (which models the interface behavior) proceeds as follows.

- The actions in the present state of the EFSM are executed "instantaneously" (i.e., zero elapsed time).
- Then, if this state is labeled with a synchronization point, control is passed to the TDI. Execution under the TDI then proceeds as explained in Section 3, with the addition of linked procedures and functions (Section 4), as well choose functions and loop predicates. The TDI execution will, in general, affect state variables of the EFSM (through calls to procedures linked to events) and allow time to advance.
- The TDI executes until a timing diagram labeled with a synchronization point has terminated its execution. Note that the TDI "remembers" all its state attributes defined in Section 3, so that the next time it regains control, it will proceed from where it left off.
- Control is then returned to the EFSM. The EFSM evaluates its state transition conditions and proceeds to the next state.

Fig. 13 shows the EFSM model for the internal behavior of the 8085 processor [13] for a small subset of four instructions: MOVM (Move

from memory), ADI (Add immediate), DCRM (Decrement memory), and CALL. Plain state transition arrows in the figure indicate that their source state is labeled with a synchronization point. Dashed arrows indicate instantaneous transitions (control is not given to the TDI). Fig. 14 shows the interface specification of the processor. In the following we illustrate the cooperative execution of the EFSM and the TDI for the DCRM instruction.

1. Initially the EFSM is in the leftmost state of Fig. 13. In this state, there is only one action: the content of the *PC* is loaded into the *Address* variable.

2. Control is then passed to the TDI. The "choose function" *CH(IFlag)* which labels the *Choice* TD in Fig. 14, chooses between a *Fetch* or an *Interrupt* machine cycle. Assuming there were no interrupt requests *IFlag = False)*, the TDI executes a *Fetch* cycle, using the *Address* value that was set by the EFSM. During the execution of this cycle, the TDI latches the data bus at the proper time and writes the data value into the *Data* variable of the EFSM (through procedural linking in the Fetch timing diagram). The Fetch timing diagram (lower left corner of Fig. 14) is labeled with a synchronization point (pictorially represented by a small dark box in the lower right corner of the Fetch TD). The TDI thus returns control to the EFSM.

3. The EFSM (Fig. 13) then evaluates the state transition conditions; these test the *Data* variable for the valid instruction opcodes; (the conditions are denoted *A, B, C and D* in Fig. 13; see bottom of figure for their precise meaning). Assume for illustration purposes, that the *C* condition evaluates to *True* (DCRM instruction). The EFSM thus moves to the next state which is enabled by the *C* condition. In this state, the EFSM sets the *Address* variable to the content of the H & L register pair (this is the address from/to which the data must be read then written back). The EFSM also sets the variables *NRead*, *NWrite*, and *NEmpty*. These indicate the number of Read machine cycles, Write machine cycles, and idle clock cycles, respectively, to be performed in the execution of the instruction. In the case of the DCRM instruction, the data must be read (*NRead* set to 1), decremented, then written back (*NWrite* set to 1); no idle cycles are needed (*NEmpty* set to 0).

4. Control is then passed to the TDI. The TDI resumes where it left off previously, i.e., at the second child of the *Concatenation* operation

labeled *Instruction Cycle* (Fig. 14). This child is a *Loop* of *NRead* iterations. The first Read (i.e., child of the *Loop* TD) is thus performed by the TDI, using the *Address* value that was set by the EFSM. During the execution of this cycle, the TDI latches the data bus at the proper time and writes that value into the *Data* variable of the EFSM (through procedural linking in the Read timing diagram). Since the Read timing diagram is labeled with a synchronization point, the TDI returns control to the EFSM.

5. The EFSM advances to the next state (the absence of state transition condition signifies a universally *True* condition). The actions in this state consist of decrementing the *Data* variable and accordingly setting the Z (zero), S (sign), P (parity) and AC (auxiliary carry) condition flags.

6. Control is then passed to the TDI. Since *NRead* was 1, *Loop(NRead)* is now over. The TDI thus resumes at the third child of the *Concatenation* operation labeled *Instruction Cycle* (Fig. 14). This child is a *Loop* of *NWrite* iterations. The first Write (i.e., child of the *Loop* TD) is thus performed by the TDI, using the *Address* value that was set by the EFSM. Since the Write timing diagram is labeled with a synchronization point, the TDI returns control to the EFSM.

7. The EFSM advances to next state. The action in this state consists of incrementing the PC by 8. Then, since this state is not labeled with a synchronization point (the outgoing is a dashed line), the EFSM performs the state transition to the next state which in this case is the initial state. The EFSM is thus ready for the next instruction.

   Since *NWrite* was 1 and *NEmpty* was 0, the next time the TDI regains control (i.e., on the next instruction), it will perform no Empty cycles, therefore the Concatenation TD labeled *Instruction Cycle* will be determined to be empty, and the TDI will resume execution at the next iteration of the top-level TD (i.e., the *Loop* labeled "8085".

The advantage of the modeling methodology illustrated above is that the interface behavior is clearly separated from the functional (internal) behavior. This allows different possibilities for the generated model. For example, to perform a high-level simulation of the 8085, the interface specification (Fig. 14) can be simply removed and replaced by atomic procedure calls ("Fetch", "Read", or "Write") which access an array data structure representing the main memory of the processor.

6 Discussion

Semiconductor and subsystem manufacturers often supply *timing diagrams* to describe the interface specifications of their products. This notation is convenient for describing signal behavior over time, and hardware designers are familiar with it. In [6], the timing diagram notation is formalized, and its expressive power extended. Looping and conditional executions of timing diagrams are supported; the control-flow of these executions is captured with "extended boolean expressions" on signals; in addition to the standard boolean connectives, these expressions include signal *Delay* and *Latch* constructs to capture state information across timing diagrams. Timing diagrams can also be put in *concurrency*; synchronization and timing constraints can be expressed between concurrent timing diagrams. In comparison, our approach to capturing interface specifications is quite similar. In [6], the specifications are used for the synthesis of interface circuits, whereas we use the specifications to generate executable (simulation) models.

In [12], the specification methodology is based on the separation of interface specifications (which are captured as in [6]) from internal data-flow specifications (captured with a textual HDL program). The approach is suitable for entities for which the overall control-flow follows closely the interface control-flow. From a specification point of view, the two descriptions are related only through I/O signal names and symbolic data names (i.e., common name space between the two specifications for I/O signals and symbolic values on data busses). As a result, the HDL specification contains control-flow information which could be redundant (e.g., Fig. 1 in [12]) with respect to that captured in the interface specification. In comparison, our approach which is based on directly linking data-flow operations to interface events ("procedural linking"), avoids this redundancy.

In [14], a VHDL annotation language, *VAL+*, is proposed to describe parametrized, hierarchical event patterns. The patterns are used for matching simulation traces; the idea is to transform (flat) simulation traces into hierarchical ones, by pattern matching, in order to help the user in trace debugging and browsing. However, the matching is done off-line, after the simulation has completed; this requires the storage of the complete simulation trace. This also implies that the approach of [14] cannot be used for checking state assertions of the modeled entity (since such checks require knowledge of the execution context). Furthermore, the VAL+ patterns are used only for trace matching, not for driving the circuit under simulation. In comparison, our *TDI* approach consists of on-the-fly

hierarchical matching; the complete simulation trace is not stored, instead only the most recent trace history is kept (under user control). Our hierarchical patterns are used for both driving simulated entities and matching their responses. Our on-the-fly matching technique, coupled with procedural linking, forms the basis for specifying state assertions to be checked during simulation. Finally, by acting on the simulated model itself (rather than just on its stimulus/response specifications), our approach also allows to conveniently carry simulations at different levels of abstraction, e.g., the generated models can perform atomic operations that stand for complete patterns of lower level events.

The *HIDE* system [15] generates VHDL interface models from timing diagrams and state diagrams. The state diagrams specify interface control-flow, similarly to our *choose functions* and a *loop predicates* in CHOICE and LOOP, respectively. A VHDL procedure is generated for each interface operation (such as READ, WRITE etc.). The procedures can then be called from a *command file* to simulate the interface behavior. This approach, however, does not seem to be practical for cases such as memory devices, wherein the choice of the actual interface operation cannot be decided before-hand (i.e., the interface control-flow is governed by the environment, e.g., the processor). Moreover, HIDE does not support hierarchical TD compositions, and its timing specification method does not support cases such as that illustrated in Fig. 3.

7 Conclusion

We have presented a modeling methodology and tool set for the rapid development of executable HDL models. The method is based on the separate capture of interface specifications, functional specifications and the relation between them. HDL models are generated in a *layered* fashion, at different levels of abstraction, in which layers can be easily inserted and removed, thus facilitating the validation of different aspects of the design. *HDL interface models* are automatically generated from the specifications.

In the future, we intend to perform additional case studies and extend the modeling methodology, e.g., to pipelined architectures. We also intend to improve the usefulness of the timing diagram interpreter, e.g., by experimenting with error recovery schemes (presently, the TDI halts its execution when an error is propagated up to the top-level). Furthermore, we intend to implement conditional trace matching in the TDI, i.e., repeatedly "hunting" for a specific pattern pre-condition (no errors are flagged

when this pattern is not matched) and starting matching only when the pre-condition is fulfilled. This is useful in checking state assertions in the modeled entity.

References

[1] A.R. Martello and S.P. Levitan, "Temporal specification verification via causal reasoning", *Proc. 2nd ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1992.

[2] K. McMillan and D.L. Dill, "Algorithms for interface timing verification", *Proc. 2nd ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1992.

[3] F. Mavaddat and T. Gahlinger, "On deducing tight bounds from partial timing specifications", *Proc. 1st ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1990.

[4] J.A. Brzozowski, T. Gahlinger, and F. Mavaddat, *Consistency and satisfiability of waveform timing specifications*, Research Report CS-88-24, University of Waterloo, 1988.

[5] S.K. Sherman, "Algorithms for timing requirement analysis and generation", *ACM/IEEE Proc. 25th DAC*, pp. 724–727, 1988.

[6] G. Borriello, *A New Interface Specification Methodology and its Application to Transducer Synthesis*, PhD thesis, University of California, Berkeley, 1988.

[7] IEEE, *IEEE Standard 1076-1987, VHDL Language Reference Manual*, IEEE, 1987.

[8] K. Khordoc, M. Dufresne, and E. Cerny, "A stimulus/response system based on hierarchical timing diagrams", *IEEE Proc. ICCAD-91*, pages 358–361, 1991.

[9] K. Khordoc, E. Cerny, and M. Dufresne, "Modeling and execution of timing diagrams with optional and multi-match events", *Proc. 2nd ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1992.

[10] K. Khordoc, M. Dufresne, and E. Cerny, "A stimulus/response system based on hierarchical timing diagrams" Publication 770, Dept. I.R.O., Université de Montréal, 1991.

[11] Texas Instruments Incorporated, *Supplement to MOS Memory Data Book*, Texas Instruments, Houston, Texas, 1984.

[12] G. Borriello, "Combining event and data-flow graphs in behavioral synthesis", *IEEE Proc. ICCAD-88*, pp. 56–59, 1988.

[13] Intel Corporation, *MCS-85 User's Manual*, Intel, Santa Clara, CA, 1978.

[14] B.A. Gennart and D.C. Luckham, "Validating discrete event simulations using event pattern mappings", *ACM/IEEE Proc. 29th DAC*, pp. 414–419, 1992.

[15] Y.H. Leong and W.P. Birmingham, "The automatic generation of bus-interface models", *ACM/IEEE Proc. 29th DAC*, pp. 634–637, 1992.

```
method update(class: leaf) (self, event)
  spec_event := current_event(signal(event)))
  match_error(self) := value_mismatch(spec_event, event)
                       OR time_mismatch(spec_event, event);
  update_longest_paths(spec_event, time(event));
  if (not (is_multi_match(spec_event) and is_input(spec_event)))
    then set current_event(sig) to next(spec_event);
  end if;

 /* validate projected events in self: */
  loop for sig in (signals(self) - signal(event)) do
    loop for spec_event from current_event_of_signal(sig)
         then next(spec_event) do
      if (sup(spec_event) < time(event))
        then match_error(self) :=
          match_error(self) or value_mismatch(spec_event, sig);
      else  /* advance current_event beyond projected events */
        set current_event(sig) to spec_event;
        exit;  /* projection completed for sig */
      end if;
    end loop;
  end loop;

  /* self is projectable if all its unoccurred events
     are projectable */
  projectable(self) := true;  /* until proven false */
  loop for sig in signals(self) do
    loop for spec_event from current_event_of_signal(sig)
         then next(spec_event) do
      if  value_mismatch(spec_event, sig))
        then   projectable(self) := false;
               exit;
      end if;
    end loop;
    if projectable(self) := false then exit; end if;
  end loop;
end update(class: leaf).
```

Figure 7: Leaf update method.

```
method update(class: choice) (self, event)
  match_error(self) := true; projectable(self) := false;
  cur_events(self) := nil;
  loop for each child in active_children(self) do
    update(child, event);
    match_error(self) :=
                    match_error(self) and match_error(child);
    projectable(self) :=
                    projectable(self) OR projectable(child);
    cur_events(self) := APPEND cur_events(child)
                                TO cur_events(self);
    if match_error(child)
      then  REMOVE child FROM active_children(self);
    elseif empty(child)
      then  set cur_events(self) to NIL;
          /* Choice has successfully terminated */
            exit; /* no need to continue loop */
    end if;
  end loop;
end update(class: choice).
```

Figure 8: Choice update method.

```
method update(class: concurrency) (self, event)
  child := the child in children(self) which is
           defined over signal(event)
  if empty(child)   /* i.e., the child defined over the signal
                       had already terminated */
    then match_error(self) := true
  else
    update(child, event);
    match_error(self) := match_error(child);
    projectable(self) :=
          (all child in children(self) are projectable(child));
    cur_events(self) := APPEND together the cur_events
                                of all children(self);
    end if;
  end if;
end update(class: concurrency).
```

Figure 9: Concurrency update method.

```
method update(class: concatenation) (self, event)
  if (projectable(current_child(self))
      and next_child(current_child(self)) /= nil
      and signal(event) has a spec event in
          cur_events(next_child(current_child(self))))
    then   /* project current_child(self) into the past: */
    SET current_child(self) TO next_child(current_child(self));
    update(current_child(self), event);
    match_error(self) :=  match_error(current_child(self));

  elseif signal(event) has a spec event in
                               cur_events(current_child(self))
    then
    update(current_child(self), event);
    match_error(self) :=  match_error(current_child(self));

  else /* illegal attempt to project */
    match_error(self) := true;
  end if;

  if empty(current_child(self))
    then SET current_child(self)
             TO next_child(current_child(self));
  end if;

  projectable(self) :=  (is_last_child(current_child(self))
                        AND projectable(current_child(self)));
  if (projectable(current_child(self))
      and next_child(current_child(self)) /= nil)
    then
    cur_events(self) := extend_cur_events(current_child(self),
                                next_child(current_child(self)));
  else
    cur_events(self) := cur_events(current_child(self));
  end if;
end update(class: concatenation).
```

Figure 10: Concatenation update method.

```
process hierarchical_timing_diagram_interpreter(TD)
  initialize(TD); output_event = nil;
  repeat
    current_time := get_current_simulation_time();
    actual_events := get_input_events_from_simulator();
    if (output_event /= NIL)
       and (current_time = time(output_event))
      then
      occur_now(output_event);
      append output_event to actual_events;
    end if;
    append_time_out_events(actual_events);
    loop for each event in actual_events do
      update(TD, event);
      if match_error(TD)
        then
        error_message(event);
        exit;
      end if;
    end loop;
    output_event := compute_output_event(cur_events(TD));
    timeout := min(time(output_event),
                   smallest_sup(cur_events(TD)))
               - current_time;
    wait on signals(input_event_subset(cur_events(TD)))
         for timeout;
  until empty(TD) or match_error(TD);
end hierarchical_timing_diagram_interpreter.
```

Figure 11: The TDI process.

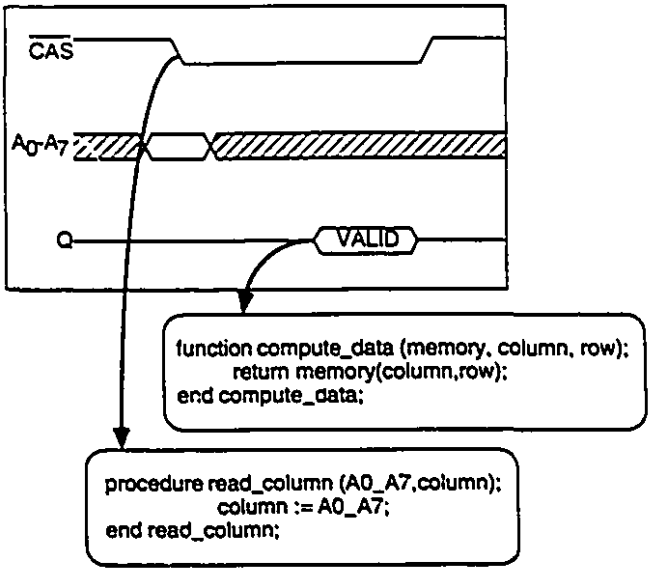


Figure 12: Example of procedure binding in VRAM.

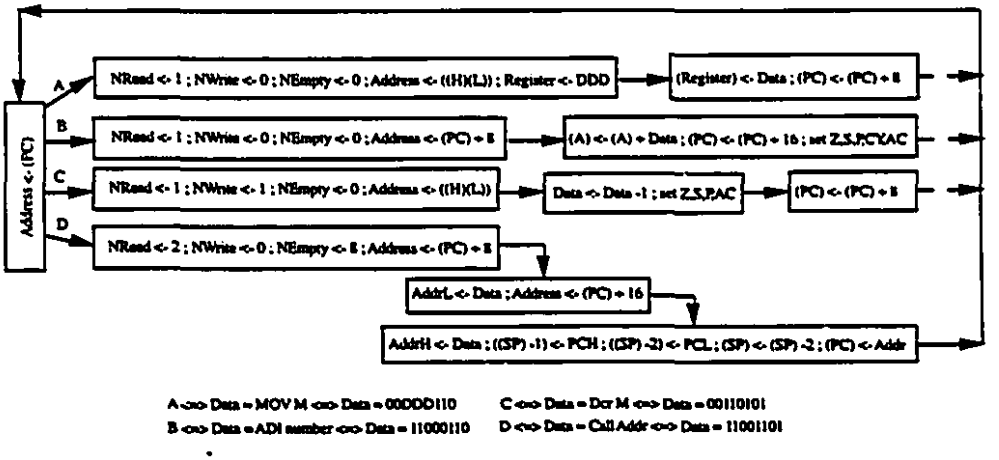


Figure 13: Excerpt from the 8085 internal behavior specification.

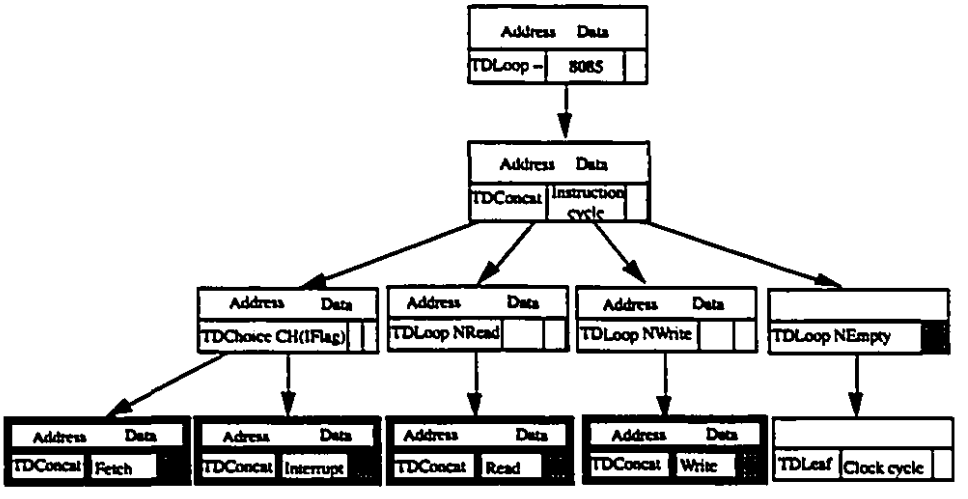


Figure 14: Interface specification of the 8085.

CHAPTER 5

MODELING CELL PROCESSING HARDWARE WITH ACTION DIAGRAMS

ABSTRACT

In this paper we address the behavioral modeling of cell processing hardware (e.g., packet / ATM switching systems). We propose a modeling methodology, *Action Diagrams*, in which the timing and protocol aspects are specified in a nearly "orthogonal" way to the data manipulation aspects, while maintaining the links between the two. We show the novel aspects of this specification paradigm and we illustrate its use on cell processing applications.

1 Introduction

When designing complex hardware systems consisting of multiple ASICs, the high level design must be verified before it is refined into an RTL design. Therefore behavioral models of the system must be developed. Although it is generally accepted that the next step in raising design productivity and reducing time to market of large systems resides in behavioral modeling, there is strong reluctance in the industrial design community to adopt behavioral modeling. This is due to the lack of established behavioral modeling methodologies. Such methodologies are bound to be application dependent (as opposed to, for example, RTL modeling methodologies which, to a great extent, are application independent).

In this paper we address behavioral modeling issues for hardware systems in packet or ATM (Asynchronous Transfer Mode) switching applications. We designate this class of systems generically as "cell processing hardware" (wherein a *cell* is a packet or an ATM cell). These applications are characterized by:

- A balanced mix of protocol aspects and data computation aspects. The protocol aspects consist, for example, of flow control mechanisms, merging and synchronization of different cell streams, processing of the handshake information embedded in the cells and the effects of this processing on the cell flow through the system. These aspects have the advantage that they can be validated independently of the payload (data) carried by the cells (which therefore facilitates the validation). A major difficulty, however, is that these protocol aspects span the system as a whole, and therefore cannot be validated locally. The data computation aspects, on the other hand, consist, for example, of algorithmic descriptions of CRC (Cyclic Redundancy Checks) and other error checking codes, etc. These data computation aspects can be typically validated locally and independently of the overall cell flow in the system.

- Real-time requirements. For example, when exploring different ATM switch architectures, latencies in the system are a concern for CBR (Constant Bit Rate) traffic, e.g., voice traffic. It is therefore important to capture timing information and timing constraints at the behavioral level.

Behavioral modeling approaches, such as [1, 2], lack the timing constraint constructs and the capability of declaring the assumptions that a

behavior makes on its environment. We found that such constructs and capabilities are important in validating the protocol aspects of cell processing applications.

Interface modeling approaches such as [3, 4, 5], are adequate when interfaces are completely defined down to physical ports and true timing. However, in the design methodology that we are considering, behavioral models must be developed well before the interfaces between ASICs are specified in detail. Furthermore, the detailed interfaces, when they are specified, are too low-level to reveal the important characteristics of system interactions in a manner that would be amenable to validation of the protocol aspects of the system.

In this paper, we propose a behavioral modeling methodology in which the timing and protocol aspects are specified in a nearly "orthogonal" way to the data manipulation aspects, while maintaining the links between the two. We show the novel aspects of the specification paradigm and we illustrate its use on cell processing applications. The methodology is based on *Action Diagrams*, which is an extension of the Timing Diagrams of [3] and [4]. In comparison to [3, 4], we have introduced important modeling concepts suitable for behavioral level modeling:

- A true behavioral hierarchy with port mappings, parameters and local variables in Action Diagrams.
- Message-based and value-based ports.
- Choice semantics supporting both deterministic and non-deterministic choice.
- An exception handling mechanism.
- A powerful functional annotation mechanism for data computation aspects.
- User-defined data types for ports and actions.
- Separation of timing constraints into *assume* and *commit* constraints.
- Timing constraint composition operations for multiple causal predecessors of an action: latest, earliest and conjunctive composition.

We have implemented a specification capture system based on Action Diagrams, and we are now implementing a model generator which produces

a behavioral VHDL model from Action Diagram specifications. We are also performing modeling experiments on industrial applications.

The rest of this paper is structured as follows: In Section 2 we overview the Action Diagrams specification method. In Sections 3 and 4 we illustrate the method on cell processing applications. Finally, Section 5 concludes the paper by discussing some future orientations of our work.

2 Action Diagrams

An Action Diagram specification represents the behavior of a system as a behavioral hierarchy. Leaf Action Diagrams and their annotated extension (for functional specifications) are presented in Section 2.1 and Section 2.2, respectively. Hierarchical Action Diagrams and their annotated extension are presented in Section 2.3 and Section 2.4, respectively.

2.1 Leaf Action Diagrams

We informally introduce the essential features of Leaf Action Diagrams by way of an example shown in Fig. 1. A Leaf Action Diagram is defined over a set of *ports*, e.g., In-port, Out-port and w-buff-full. The *type* of a port can be any VHDL compatible type. For example, In-port and Out-port are of type cell-type (a user-defined type) and w-buff-full is of type binary. Ports have a *direction*, e.g., *in*, *out*, and *inout*, for In-port, Out-port and w-buff-full, respectively. *Internal* ports can also be specified; their semantics are similar to *out* ports, except that their behavior is not visible from outside the action diagram.

The behavior of a port is captured as a sequence of *actions*. An action has a *direction*; in the case of *in*, *out* and *internal* ports, the direction is inherited from the port; in the case of *inout* ports, the direction of the action must be specified, e.g., that of the first action on w-buff-full is *out*, and the second one is *in*.

Actions are labeled. The label can be:

- A constant, or a symbol denoting a constant of the corresponding data type indicating that the port will take on that value and then remain stable. For example, the first action on w-buff-full is labeled *low*.

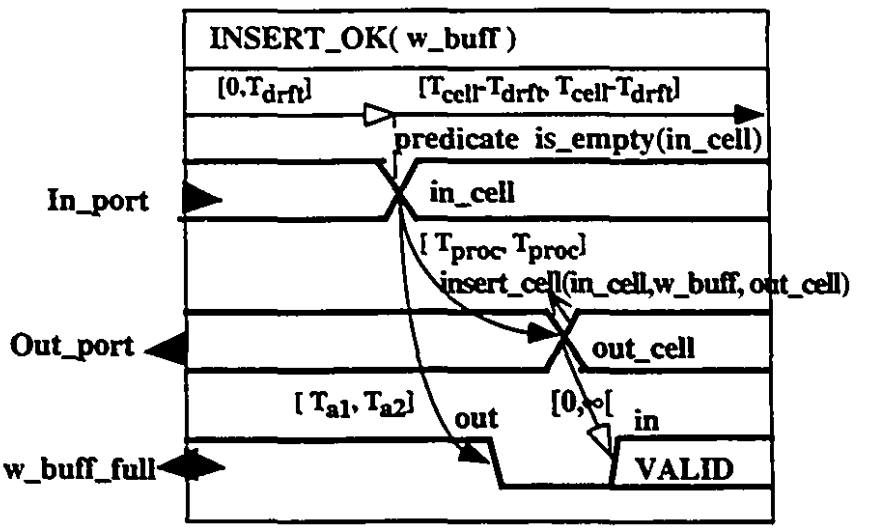


Figure 1: Example action diagram.

- The special symbol *valid*, indicating that the port will take on any value of the data type, and then remain stable. For example, the action labeled valid on w-buff-full indicates that the port can remain *low*, or become *high* (driven by the environment, as indicated by the *in* direction of the action) and then remain *high*. In the case of an output, the value *valid-val*(port) is used (the function *valid-val* must be defined for each port data type carrying *valid* labeled output actions).

- The special symbol *dont-care*, indicating arbitrary or unspecified behavior of the port. In the case of an input action, the action can match an arbitrary number (including zero) of actual action occurrences on the port. In the case of an output action, and for modeling purposes, the port is driven to the value *dont-care-val*(port). The function *dont-care-val* must be defined for each port data type carrying *dont-care-val* labeled output actions.

Initial value labels can be specified for *in* and *inout* ports. These labels are the same as action labels. When specified, they indicate what the value of the port must be when the action diagram starts. For example, w-buff-full specifies a *low* initial value, while In-port does not specify an initial value.

A port can be *value-based* or *message-based* (this is designated as the *interpretation* of the port). In the former case, the action diagram in effect declares that it expects to be notified of the occurrence of an

input action on the given port, only if that action modifies the value of the port. For example, w-buff-full is value-based. If it remains *low* after out-cell occurs (which is allowed by the *valid* action on w-buff-full), no actual input action need be received (and actually none will) in order to match the action labeled *valid*. In the latter case (message-based), action signaling is independent of action values. For example, In-port and Out-port are message-based. An action must be actually received on In-port in order to match the specified in-cell action, independently of the value of the previous cell received on the port. Similarly, out-port must be updated at each out-cell.

Virtual *Start* and *End* actions (represented by the left and right vertical boundaries of the action diagram) delimit the scope of the action diagram. The *Start* action precedes all actions and the *End* action succeeds all actions of the action diagram.

Actions can be related by weighted (min/max) *timing constraints*. Timing constraints can be of *assume* or *commit* intent, indicated by empty-headed and black-headed arrows, respectively. *Commit* timing constraints specify the order and/or timing in which output actions are generated by the action diagram. *Assume* timing constraints specify assumptions that the action diagram makes on the order and/or timing of actions. For example, there is an *assume* timing constraint of weight $[0, T_{drft}]$ from the *Start* action to the in-cell action on In-port. This indicates that a cell must be received on In-port within a delay T_{drft} from the beginning (*Start* action) of the action diagram. There is a *commit* timing constraint of weight $[T_{a1}, T_{a2}]$ from the action on In-port to the action on w-buff-full, indicating that, when a cell is received on In-port, the w-buff-full signal will be driven *low* after a minimum delay T_{a1}, and a maximum delay T_{a2}.

Consider a set $S = \{C_1, \ldots, C_n\}$ of timing constraints, such that the elements of S are all of the same intent (*commit* or *assume*) and are all incident on the same action E (Fig. 2). The interpretation of the timing constraints can be one of three kinds (specified by the user): conjunctive (all predecessors determine the occurrence of the action), earliest or latest (only the earliest or the latest arriving predecessor determines the occurrence of the action). More precisely, let E_i (resp. $[l_i, u_i]$) be the source action (resp. weight) of constraint C_i, and let t_i be the occurrence time of E_i, $i = 1 \ldots n$. Then t, the occurrence time of E, is as follows:

(a) Conjunctive$(C_1, \ldots, C_n)$: $\forall i,\ t_i + l_i \leq t \leq t_i + u_i$
(b) Latest$(C_1, \ldots, C_n)$: $max_i(t_i + l_i) \leq t \leq max_i(t_i + u_i)$
(c) Earliest$(C_1, \ldots, C_n)$: $min_i(t_i + l_i) \leq t \leq min_i(t_i + u_i)$

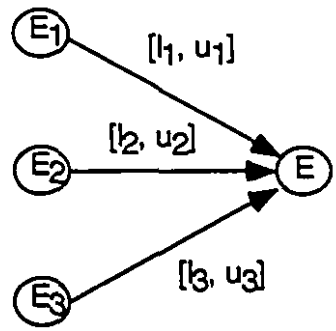


Figure 2: Multiple constraints with the same action sink.

The operational semantics of an action diagram are defined in terms of its execution in an environment which drives the diagram's *in* and *inout* ports and observes its *out* and *inout* ports. During such an execution, the action diagram is said to be in a *satisfying status* when:

- its initial value specifications are satisfied
- its specified input actions are matched, i.e.,
  - they satisfy the specified action sequences on ports,
  - they satisfy the *assume* timing constraints, and
  - they satisfy the value specifications given by the action labels.

When an action diagram takes on a non-satisfying status, it is *disabled*, i.e., its execution is terminated. The implications of this depend on the instantiation context of the action diagram; this is further elaborated in Section 2.3. If, however, the action diagram maintains a satisfying status until it fires its *End* action, we say that the action diagram *completes* (its execution).

The simple concepts explained above, such as action sequences, assume timing constraints, and action labels, lead to a natural and easy way of specifying more complex and useful properties. For example, the *low* to *valid* pattern on w-buff-full combined with the $[0, \infty]$ *assume* timing constraint from out-cell to valid w-buff-full states that *if* w-buff-full is reasserted (driven High), *then* it can happen only after out-cell has been sent out.

2.2 Annotated Leaf Diagrams

Action diagrams can have *parameters*. In Fig. 1, w-buff is an *in* parameter (of type cell-type). More generally, *out* and *inout* parameters can also be defined.

The action and initial value label set is extended to arbitrary symbols, in addition to those denoting constants introduced in Section 2.1. The semantics are the same as in the case of the *valid* label, and in addition, the symbolic label has the effect of declaring a *variable* of the corresponding data type and of local scope to the action diagram. In the case of an input action, the actual value of the port is latched into that variable. For example, the label in-cell on the action of In-port declares a variable of cell-type that will, at the occurrence of the action, be assigned the value of the cell received on the In-port. For an output action, the value of the variable is used to drive the port. For example, the label out-cell on the action of Out-port declares a variable of cell-type, whose value will be assigned to Out-port at the time of the occurrence of the action.

Additional variables, not directly related to port actions can be specified. They typically serve as place holders for the results of intermediate computation in the action diagram.

Predicates and procedures (written in VHDL), having as input parameters variables (which include those declared by action symbols) and/or parameters of the action diagram, can be attached to an action (in, out, internal and *Start/End* actions). These predicates and procedures are computed in "zero time" at the time instant at which the corresponding action occurs, and they must contain no reference to time, delays, nor synchronization (e.g., WAIT statements). Predicates extend in a natural way the satisfaction semantics of action diagrams. For example, the predicate is-empty attached to the action labeled in-cell on In-port in Fig. 1, has as input parameter the variable in-cell, and tests whether the cell is "empty"; if this is not the case (i.e., the cell is not empty), the action diagram is disabled. Procedures can have output and inout parameters (in addition to input parameters), and can modify the variables and parameters (out and inout) of the action diagram. For example, the procedure insert-cell attached to the action labeled out-cell in Fig. 1, takes as *in* parameters the variable in-cell and the parameter w-buff (of the action diagram). The procedure then computes a new cell, and puts the result in the variable out-cell. There can be at most one procedure attached to any given action (for more than one procedure, an additional level of procedural nesting must be used, which will then determine the correct order of execution).

The execution semantics at the time of occurrence of an action (or of multiple actions occurring at the same time) are:

1. Update all variables associated with input actions that have occurred at the current time instant.
2. Evaluate all predicates attached to actions in 1, and to *out* and *internal* actions chosen to occur at the current time instant.
3. Execute (in arbitrary order) all procedures attached to actions in 2 (*in*, *out*, and *internal*).
4. Update all ports corresponding to *out* and *internal* actions occurring at the current time instant.

2.3 Hierarchical Action Diagrams

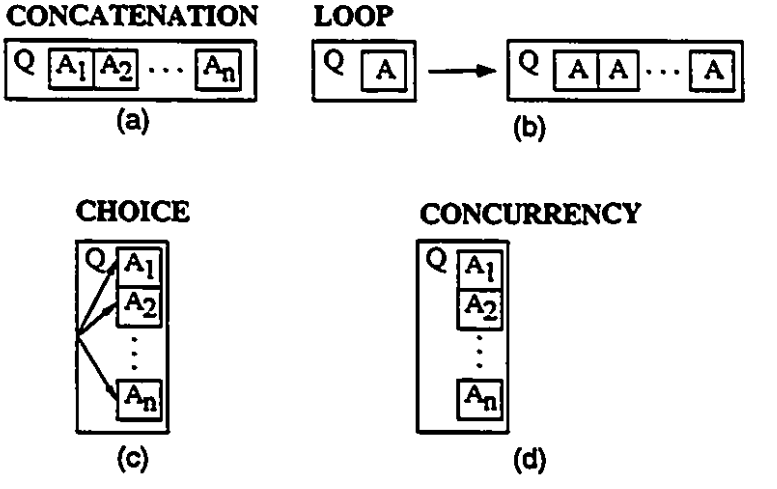


Figure 3: Action diagram composition operations.

Action diagrams can be hierarchically composed. A hierarchical action diagram Q is defined by a set of *external* ports (i.e., *in*, *out*, *inout* ports), a set of *internal* ports, an ordered list of child action diagrams $(A_1, \ldots, A_n)$, a hierarchical *composition operation*, and a *port map* for each A_i, $i = 1, \ldots, n$. The composition operations (Fig. 3) are: *Concatenation*, *Loop*, *Concurrency* and *Choice*. The port map establishes the correspondence between the external ports of A_i and the ports of Q (both external and internal).

For all operators, except *Choice*, a status of non-satisfaction in one of $A_1, \ldots, A_n$, unconditionally translates to a status of non-satisfaction for Q. In the case of *Choice*, the semantics are slightly more complex (explained below).

Concatenation: A_1 starts when Q starts. A_{i+1} starts when A_i completes. Q completes when A_n completes.

Loop: The semantics are similar to *Concatenation* with an infinite number of identical A_i's.

Concurrency: $A_1, \ldots, A_n$ start when Q starts. Q completes when all of $A_1, \ldots, A_n$ complete. When multiple A_i's write to the same port, the resulting behavior is similar to that of a multiple-writer shared variable, i.e., at all times, the value of the shared port is that of the last value written. If multiple writes occur at the same time instant, the result is unpredictable (the write actions are serialized, and the last one "wins").

Choice: $A_1, \ldots, A_n$ represent alternative (branching) behaviors. The behavior of Q is governed by concurrent choice semantics in which all of the $A_1, \ldots, A_n$ execute concurrently. $A_1, \ldots, A_n$ start when Q starts. When an A_i takes on a non-satisfying status, it is disabled (its execution is terminated). If all the A_i's take on a non-satisfying status, Q takes on a non-satisfying status as a result. Two kinds of choice are supported: deterministic and non-deterministic. The user specifies the desired kind for each usage of the *Choice* construct.

In the following, an action diagram is said to produce a *side effect* at a given time instant, if it produces an output action or executes a procedure that could modify an *out* or *inout* parameter of the action diagram at that time instant.

- *Deterministic Choice*: When a choice branch A_j produces a side effect or completes (whichever comes first), A_j must be the only still enabled branch in that *Choice* (i.e., all other branches must have had already been disabled). Otherwise, it is an error.
- *Non-Deterministic Choice*: When a choice branch A_j is about to complete or produce a side effect at the current time instant, if A_j is not the only still enabled branch in that *Choice*, a non-deterministic selection of one of the still enabled choice branches is made, and all other branches are disabled. The execution of the selected choice branch then proceeds normally.

The two *Choice* constructs thus allow a *delayed choice*, whereby the

selection of a choice branch is delayed until sufficient information is gathered. This is useful in supporting "scenario-based" modeling. Furthermore, a simple modification to the deterministic delayed choice semantics leads to the definition of an exception handling mechanism. This is further explained in Section 3.

2.4 Annotated Hierarchical Diagrams

A selection function can be optionally associated with a *Choice* action diagram Q. The *in* parameters of the function can be any subset of the input parameters of Q. The function is evaluated when the *Choice* is entered and returns a subset of m choice branches (designated as the "selected" branches) out of the n possible branches ($1 \leq m \leq n$). After this initial selection, the semantics of the *Choice* are the same as in Section 2.3.

Similarly, a loop predicate P can be associated with a *Loop* action diagram Q. The *in* parameters of the predicate can be any subset of the variables of the action diagram that contains Q. The semantics are: (WHILE P (LOOP Q)), i.e., the predicate P is evaluated before every iteration.

3 Example: a Rate Adaptation Queue

This class of queue is typical of cell processing applications. Its behavior is as follows:

- Cells arrive on a write-port (input of type cell), and are queued.
- Cells depart on a read-port at a constant rate.
- When the queue is empty, "empty" cells (cells with no real payload, and with a special identifier in the header) are output. Note that in this application, it is known that, on average, the queue input rate is slower than the output rate.
- A reset can occur at any time during system operation.

The architecture of the model is:

- Concurrent Read/Write accessors.

- A central storage shared by the accessors.
- Read & Write procedures.
- An exception handler for system Reset.

The behavior of the queue in the absence of exception conditions is given by the hierarchical action diagram QUEUE-Running (Fig. 4). Read-port and Write-port are *out* and *in* ports, respectively, both being message-based. Queue is an *inout* parameter of type queue-type (a data structure containing the actual queue object and its head and tail pointers). QUEUE-Running is composed of two concurrent infinite loops over the leaf action diagrams WRITE-A-CELL (Fig. 5) and READ-A-CELL (Fig. 6).

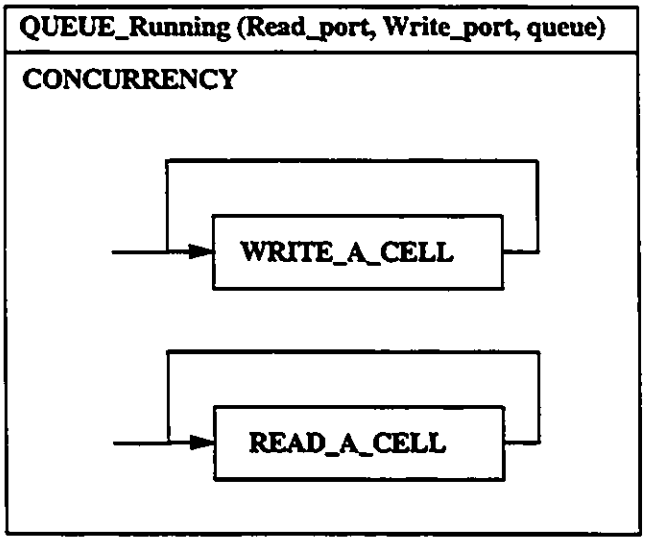


Figure 4: Action diagram for "normal" behavior of queue.

In Fig. 5, the *assume* timing constraint of weight $[T_{W_{min}}, \infty]$ declares the maximum rate at which the queue can be written into. The WRITE procedure saves cell-in (*in* parameter of WRITE) in the queue (*inout* parameter of WRITE) and updates its tail pointer. The *commit* timing constraint of weight $[0, 0]$ has the effect of ending the WRITE-A-CELL action diagram (therefore enabling the next iteration of WRITE-A-CELL), as soon as cell-in is received on the Write-port.

Cells are output from the queue at a constant rate given by the *commit* timing constraints of weight $[T_R, T_R]$ and $[0, 0]$ in Fig. 6. When the head and tail pointers of the queue coincide, The READ procedure sets

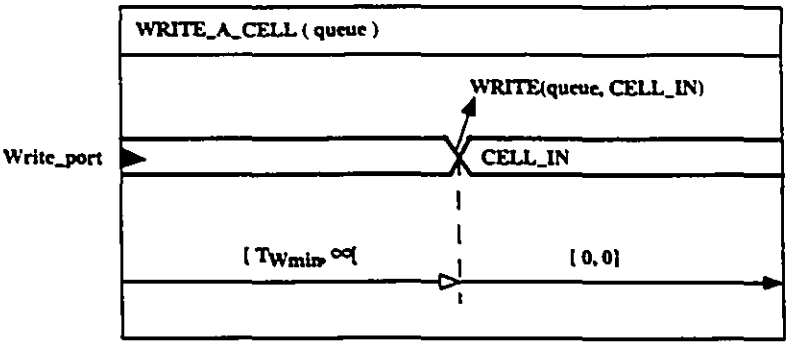


Figure 5: Queue Write action diagram.

cell-out (which is an *out* parameter of the procedure) to an empty cell. Otherwise, cell-out is set to the cell currently at the head of the queue, and the head pointer is updated. Note that cell-out was actually declared by the label on the *out* action of Read-port, and its value is thus used to drive the Read-port.

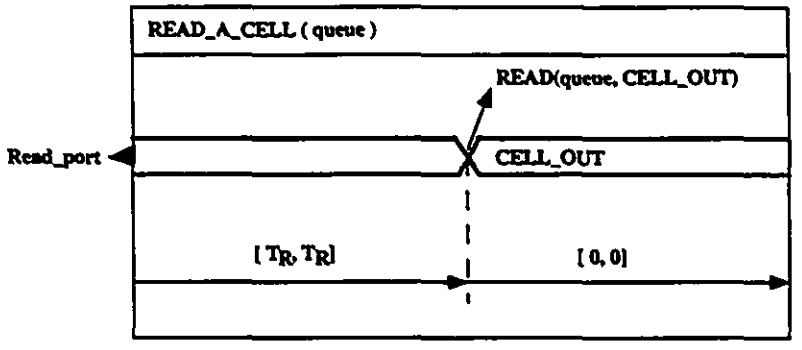


Figure 6: Queue Read action diagram.

In the following, an action diagram is said to be *passive* if it has no *out* actions, nor *out* or *inout* parameters. The *Exception-handling* operator shown in Fig. 7 implements a sufficiently general-purpose exception handling mechanism for most applications. The operator is given a normal-behavior, a *passive* exception-condition, and an exception-behavior, with all three behaviors expressed as (possibly hierarchical) action diagrams. The resulting behavior, Q, is:

- The normal-behavior and exception-condition action diagrams start when Q starts.
- If exception-condition completes before normal-behavior (and while this latter is still enabled), the execution of normal-behavior is im-

mediately terminated, and exception-behavior is executed. Q will then complete when exception-behavior completes.

- In all other cases, the behavior of Q is the same as that of normal-behavior.

```
(Exception_handling
   exception_condition
   normal_behavior
   exception_behavior)
```

Figure 7: The *Exception-handling* operator.

Using the *Exception-handling* operator, we can express the complete behavior of the queue. This is shown in Fig. 8. QUEUE-Running was defined in Fig. 4. QUEUE-RESET-START and QUEUE-RESET-DO-IT are the exception-condition and the exception-behavior, respectively (see Fig. 9). In the former, the *assume* timing constraint of weight $[0, \infty]$ expresses that the action diagram waits for a Reset for an unbounded amount of time. In the latter, the procedure INIT-QUEUE performs the initialization of the queue.

```
(defBehavior RATE_ADAPTATION_QUEUE (write_port read_port
                                                reset_port queue)
  (loop
    (Exception_handling
      (QUEUE_RESET_START Reset_port queue)
      (QUEUE_RUNNING Write_port Read_port queue)
      (QUEUE_RESET_DO_IT Reset_port queue))))
```

Figure 8: Rate adaptation queue.

4 Example: Auxiliary Cell Insertion

A Cell Flow Processor (Fig. 10) accepts cells on its In-port, processes them, and then outputs them on Out-port. There are empty cells in the cell traffic carried by In-port. The cell flow processor takes advantage

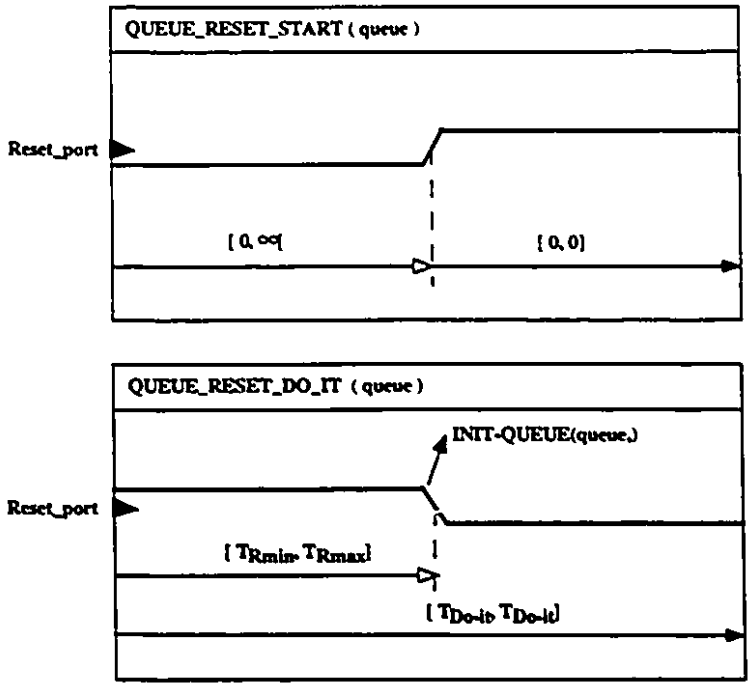


Figure 9: Exception condition (QUEUE-RESET-START) and exception behavior (QUEUE-RESET-DO-IT).

of these empty cell opportunities to insert cells from an auxiliary source (Aux-source in Fig. 10) into the outgoing cell traffic on Out-port. The cell flow processor has an internal buffer (w-buff in Fig. 10) to hold one auxiliary cell while it is waiting to be inserted in the outgoing cell flow. The Aux-source communicates with the cell flow processor through the Write-Interface of the cell flow processor. This interface consists of two ports: Aux-port and w-buff-full. The Aux-source is allowed to submit a cell to the cell flow processor (on Aux-port) only if w-buff-full is de-asserted (*low*). When the Write-interface receives a cell on the Aux-port, it asserts w-buff-full and stores the cell in w-buff. Eventually, the cell-flow unit of the cell flow processor will insert the auxiliary cell in the outgoing traffic on Out-port. It will then de-asserts (*low*) w-buff-full.

The action diagram model of the cell flow processor is shown in Fig. 11. It consists of a local variable w-buff of cell-type and a *Concurrency* over a Write-interface action diagram and a Cell-flow action diagram. In-port and Aux-port are *in* ports of the cell-flow-processor; w-buff-full and Out-port are its *out* ports. w-buff-full is value-based. The other ports are message-based.

The Write-interface action diagram is shown in Fig. 12. It consists of

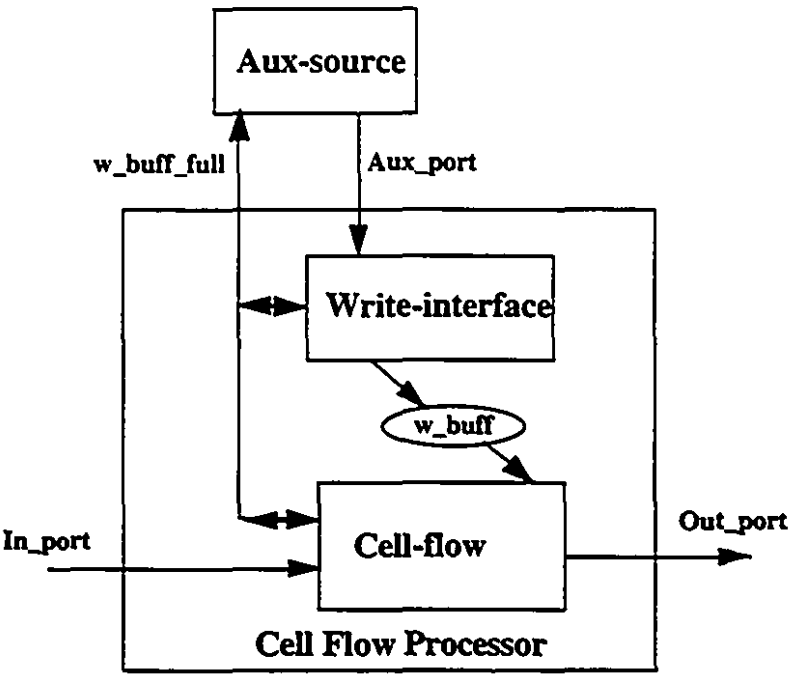


Figure 10: Example of auxiliary cell insertion.

a *Loop* over the leaf action diagram CELL-WRITE. CELL-WRITE has an *out* parameter w-buff, an *in* port Aux-port, and an *inout* port w-buff-full. The action pattern on Aux-port and w-buff-full, and the timing constraint from the first action on Aux-port to the first action on w-buff-full, specify that a cell must not be received on Aux-port unless w-buff-full is *low*. When a cell is received on Aux-port, the CELL-WRITE action diagram stores the cell in w-buff after some processing (with the procedure store-in-w-buff) and asserts w-buff-full (*high*) after a minimum delay of T_{ack1} and a maximum delay of T_{ack2}. Note that if no data processing were needed

```
(defBehavior cell_flow_processor (In_port Out_port
                                  Aux_port w_buff_full)
  (var w_buff (type cell_type))
  (concurrency
    (Write_interface Aux_port w_buff_full w_buff)
    (Cell_flow In_port Out_port w_buff_full w_buff)))
```

Figure 11: Action diagram for Cell Flow Processor.

on the cell, we could do without the procedure store-in-w-buff, and simply label the action on Aux-port with w-buff. The *assume* timing constraint of weight $[1, T_{ins,max}]$ from the assertion of w-buff-full to its subsequent de-assertion, declares a requirement that the auxiliary cell must be inserted, and thus w-buff-full de-asserted (by an *in* action), within a maximum of $T_{ins,max}$ time.

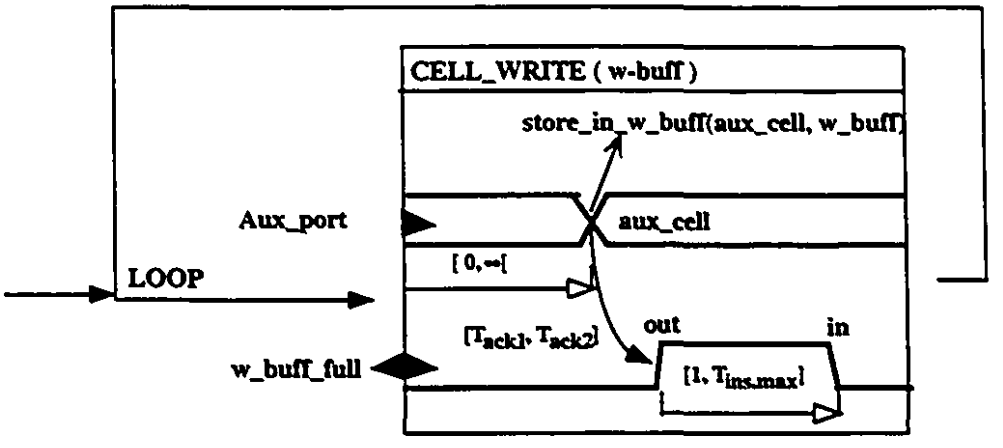


Figure 12: Action diagram for Write Interface of Cell Flow Processor.

The Cell-flow action diagram is shown in Fig. 13. It has an *in* parameter, w-buff, an *in* port, In-port, and *out* port, Out-port, and an *inout* port, w-buff-full. It consists of a *Loop* over a *deterministic Choice* of three leaf action diagrams: NOTHING-TO-INSERT, INSERT-OK and UNABLE-TO-INSERT, with *in* parameter w-buff. w-buff-full is an *inout* port of the action diagram INSERT-OK and an *in* port of the action diagrams NOTHING-TO-INSERT and UNABLE-TO-INSERT.

NOTHING-TO-INSERT corresponds to the case when w-buff-full is *low* when a cell (in-cell) is received on In-port. This cell is processed (procedure process-cell) and sent on the Out-port after a delay of T_{proc}. The *assume* timing constraint from in-cell to the *valid* action on w-buff-full indicates that the latter might possibly be asserted after the reception of in-cell on In-port.

INSERT-OK corresponds to the case when there is a cell to insert (w-buff-full is *high* on reception of in-cell) and there is an insertion opportunity because in-cell is empty. The INSERT-OK action diagram remains enabled at the reception of the in-cell action only if the predicate is-empty attached to this action returns true. If this is the case, the cell in w-buff is processed (procedure insert-cell), then sent on the Out-port after a delay of T_{proc}, and w-buff-full is de-asserted *low* after a minimum delay T_{a1}, and a maximum

delay T_{a2}. Subsequently, w-buff-full is allowed to be re-asserted (*in* action labeled *valid* on w-buff-full).

UNABLE-TO-INSERT corresponds to the case when there is a cell to insert, but there is no insertion opportunity (is-not-empty(in-cell)). In this case, w-buff-full must remain *high* (this checks whether, e.g., the Write-interface erroneously de-asserts w-buff-full). Finally, in-cell is processed (procedure process-cell) and sent on the Out-port after a delay of T_{proc}.

5 Conclusion

We have proposed a behavioral modeling methodology in which the timing and protocol aspects are specified in a nearly "orthogonal" way to the data manipulation aspects, while maintaining the links between the two. We have shown how this methodology can be applied to the behavioral modeling of cell processing hardware applications. In the future, we plan to define (for these applications) a classification of behavioral models into levels of abstraction and a formalization of the refinement steps between the different levels. We also plan to explore the re-use of high level models in the validation of lower-level models, e.g., by using action diagrams to express the relations between the levels.

References

[1] S. Narayan, F. Vahid and D. Gajski, "System Specification and Synthesis with the SpecCharts Language", *IEEE Proc. ICCAD-91*, 1991.

[2] D. Drusinsky and D. Harel, "Using StateCharts for Hardware Description and Synthesis", in *IEEE Transactions on Computer-Aided Design*, 1989.

[3] G. Borriello, *A New Interface Specification Methodology and its Application to Transducer Synthesis*, PhD thesis, University of California, Berkeley, 1988.

[4] K. Khordoc, M. Dufresne, E. Cerny, P.A. Babkine and A. Silburt, "Integrating Behavior and Timing in Executable Specifications", in *IFIP Conference on Hardware Description Languages and their Applications (CHDL)*, 1993.

[5] Y.H. Leong and W.P. Birmingham, "The Automatic Generation of Bus-Interface models", in *ACM/IEEE Proc. 29th DAC*, pp. 634–637, 1992.

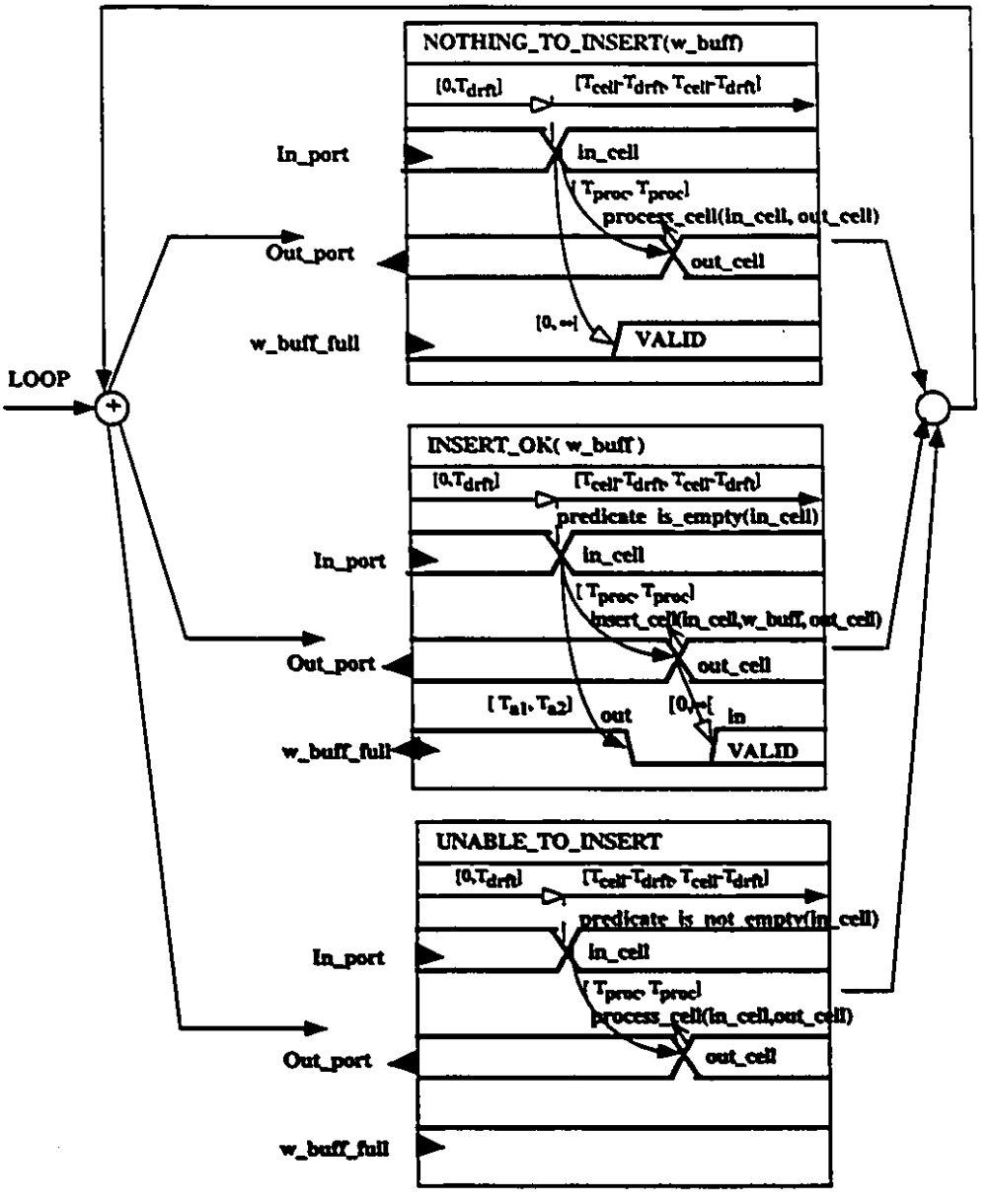


Figure 13: Action diagram for Cell Flow unit.

CHAPTER 6

SEMANTICS AND VERIFICATION OF ACTION DIAGRAMS WITH LINEAR TIMING CONSTRAINTS

ABSTRACT

Specifications containing linear timing constraints, such as found in action diagrams (timing diagrams) defining interface behaviors, are often used in practice. Although efficient $O(n^3)$ shortest path algorithms exist for computing the minimum and maximum time distances between actions, subject to the timing constraints, there is so far no accurate method that can decide a) whether a specification of this kind is realizable (i.e., can be simulated by a causal system), and b) given the action diagrams of the interfaces of two or more communicating systems, whether the systems implementing such independent specifications will correctly interoperate (i.e., satisfy the respective protocols and timing assumptions). First we illustrate the weaknesses of existing action diagram verification techniques: the causality issue is not addressed, and the proposed methods to answer the compatibility (interoperability) question yield false negative answers in many practical situations. We then define the meaning of causality in an action diagram specification and state a set of sufficient conditions for causality to hold. This development then leads to an exact procedure for the verification of the interface compatibility of communicating action diagrams. The results are illustrated on a practical example.

1 Introduction

Methods have been developed for the synthesis of interface controllers [Borr88] and for the verification of interface compatibility [Brzo91] of communicating systems described by action diagram specifications (also called timing diagrams). Other works address the issue of efficient algorithms for computing the maximal time distances between actions for more complex forms of timing constraints in action diagrams [MacM92, Burk93], or for cyclic (process like) action diagrams [Amon93] defined using the *latest* timing constraints only. However, none of these methods address the issue of realizability of such specifications in the sense of causality (i.e., can the specification be simulated by a causal system), especially in the presence of conjunctive *linear* constraints. Due to their declarative style (as opposed to, e.g., an operational style), these constraints make the causality issue a non-trivial one. In practice, synthesis methods such as [Borr88] that do not examine the causality issue under linear constraints, may produce systems that only satisfy mutually incompatible subspaces of their respective specifications. The consequence is the risk of incompatibility between independently developed implementations of the interacting systems. In [Ku92], the authors define a realizability criterion called *well-posedness*, which can be seen as a special case of our causality criterion. Well-posedness is not sufficiently powerful for reasoning on some of the practical examples that we examined (e.g., interface operations of a Motorola MC68360 processor). Recently, timed process algebras have emerged [Klus93] in which the occurrence times of actions can be related by linear conjunctive constraints. However, the underlying semantic models proposed in these works do not address the causality issue. Hence, such methods do not reveal whether the specified system can be built from independently developed subsystems, each constructed according to its local specification.

The paper is structured as follows. In Section 2, we introduce some basic concepts and notation. In Section 3, we show that known compatibility verification methods, e.g., [Brzo91], can yield *false negatives* in practical situations. This is because these methods do not *compose* the interface behaviors of the communicating systems. We show that such composition must encompass the concept of *realizability*, or else the compatibility question can yield *false positives*. We then develop, in Sections 4 and 5, formal operational semantics of action diagrams under linear timing constraints. The semantics are based on the derivation, from the action diagram, of a *block machine*. Such a

machine is characterized by a partition of the action set of the action diagram. In Section 6, we formally define the concept of a *causal* block machine, and then state the realizability of an action diagram specification in terms of the existence of a causal block machine derived from the action diagram (the derivation is defined given a partition of the action set of the action diagram, however the computation of the actual action partition is outside the scope of this paper). We then propose, in Sections 8 and 9, a set of provably sufficient (and computable) conditions for a block machine to be causal. This allows us to write an exact procedure for determining whether a block machine is causal. In Section 10, we prove that all causal block machines derived from an action diagram have the same (timed) trace set and this trace set is equal to that of the action diagram. in Section 11, we define the compatibility of communicating causal action diagrams in terms of the compatibility of *all* the combinations of causal block machines derived from these action diagrams. We prove that we do *not* need to enumerate these combinations to answer the action diagram compatibility question. This leads to an exact and efficient procedure for the verification of the compatibility of communicating action diagrams. Finally, in Section 12 we prove that the structure of the partition of the set of input actions of a causal block machine is independent of that of its output actions. In addition to being intuitively "reassuring", this property should be useful in designing an efficient action partitioning procedure.

2 Action Diagrams

An action diagram (AD) specifies, in a declarative manner, the action based, transactional aspect of a finite excerpt of the interface behavior of a system. This specification comprises the actions of the system itself (its "commitments") , as well as its assumptions on the actions that the environment can produce. Actions occur on "ports", in a punctual, instantaneous manner. An action a_k has a ***time stamp*** variable denoted by $t(a_k)$. Time stamps take on finite, possibly unbounded, real values.

Definition 1 [***Intervals*** and ***Timing Constraints***] An interval π is a set of real numbers. The interval is represented by its lower and upper bounds, T_{min} and T_{max} respectively, where $T_{min} \in Z \cup \{-\infty\}$, $T_{max} \in Z \cup \{\infty\}$, $T_{min} \leq T_{max}$,

and where $\mathcal{Z}$ designates the set of rational numbers. Such an interval π is the subset of all real numbers such that, for any t in π, t is finite (but possibly unbounded), and:

1. $T_{min} \leq t \leq T_{max}$ if T_{min} and T_{max} are both finite (π is denoted by $[T_{min}, T_{max}]$).

2. $T_{min} \leq t < \infty$ if T_{min} is finite and $T_{max} = \infty$ (π is denoted by $[T_{min}, \infty)$).

3. $-\infty < t \leq T_{max}$ if $T_{min} = -\infty$ and T_{max} is finite (π is denoted by $(-\infty, T_{max}]$).

4. $-\infty < t < \infty$ if $T_{min} = -\infty$ and $T_{max} = \infty$ (π is denoted by $(-\infty, \infty)$).

The set of intervals, denoted I, is partitioned into the subsets I_{conc} of ***concurrency*** intervals and I_{prec} of ***precedence*** intervals. The elements of I_{conc} are characterized by $T_{min} \leq 0$ and $T_{max} \geq 0$, and the elements of I_{prec} by $T_{min} > 0$. A ***timing constraint*** is a triplet $c = (a_i, a_j, \pi)$ where a_i and a_j are actions such that $a_i \neq a_j$, and π is an interval. The arithmetic semantics of the constraint are given by substituting the term $t(a_j) - t(a_i)$ for t in the appropriate item in 1 to 4 above. The resulting pair of inequalities is the ***proposition associated with the constraint*** c. A constraint with a precedence (concurrency) interval is a precedence (concurrency) constraint.

❑

Definition 2 [***Action Diagram***] An action diagram is the tuple AD = ($\mathcal{S}$, $\mathcal{A}$, o, C), where:

- $\mathcal{S}$ is a set of ***ports***. A port has a ***direction*** (*in* or *out*) and a sequence of actions. The action sets of any two ports of $\mathcal{S}$ are non-intersecting.
- $\mathcal{A}$ is a set of ***actions*** such that $\mathcal{A} = \mathcal{A}' + \{o\}$, where $\mathcal{A}'$ is the union of the action sets of $\mathcal{S}$.
- o, the ***origin action***, is a special action that marks the time at which the action diagram starts "executing". This action does not correspond to any real action of the modeled system.
- An action has a ***direction***. The direction of o is the *null* direction. The direction of an action of $\mathcal{A}'$ is that of the port to which it belongs.
- C is a set of ***timing constraints*** such that $C = C' \cup C^0$, where:
  - C' is a relation on $\mathcal{A}' \times \mathcal{A}' \times I$, where I is the set of real intervals given in Definition 1.

- $C^0 = \{(o, a_{1i}, [\varepsilon, \infty)) \mid \exists\, s_i \in \mathcal{S}, a_{1i} = first(s_i)\}$ where ε is an arbitrarily small[1] positive rational, and $first(s_i)$ is the first action in the sequence of actions of a port s_i.

- ***Restriction***: any constraint (a_i, a_j, π), in which a_i and a_j are of different directions, must be a precedence constraint, i.e., $\pi \in I_{prec}$.
- ***Constraint intent***: a constraint (a_i, a_j, π) is considered to have an *assume* (*commit*) *intent* if the direction of action a_j is *in* (*out*). The semantics of assume constraints, from a synthesis point of view, are that the designer of the system can safely assume that these constraints will hold (and he/she can take advantage of these assumptions in the design of the system). The semantics of commit constraints are that the designed system must satisfy them.

□

In the graphical representation of action diagrams, an action is represented by a short vertical bar (e.g., Figure 1), or by a circle (e.g., Figure 5 on page 58). Actions on the same port are horizontally aligned. The action sequence of a port is shown in left-to-right order. A constraint (a_i, a_j, π) is represented by an arrow labeled with the interval π and pointing from a_i to a_j. The constraint arrowhead is hollow (filled) for assume (commit) constraints.

Definition 3 [***Trace over an action set*** $\mathcal{A}$] A trace over an action set $\mathcal{A}$ is a sequence $\Omega = [\omega_i]$, $i = 1,\ldots, \theta$, where, for any i, $i = 1,\ldots, \theta$, ω_i is a set of j_i pairs such that (where $\Re$ designates the set of finite real numbers):

$$\omega_i = \{ (a_{ij}, \tau_i) \mid j = 1,\ldots, j_i, a_{ij} \in \mathcal{A}, \tau_i \in \Re \}$$

and such that $\tau_i < \tau_{i+1}$, $i = 1,\ldots, \theta - 1$. If each action of $\mathcal{A}$ appears *at most* once in Ω, we say that Ω is a ***well-behaved trace***. If each action of $\mathcal{A}$ appears *exactly* once in Ω, Ω is a ***complete trace*** over $\mathcal{A}$. A well-behaved trace that is *not com-*

1. From the point of view of the implementation of CAD software, each interval bound could be conveniently qualified by a boolean attribute indicating whether the corresponding inequality is strict or non-strict (e.g., as is done in [Dill89]), and thus an ε lower bound would be actually represented as a strict 0 lower bound. Bound comparisons and shortest paths algorithms (which we use in the resolution of timing constraint systems) can then be easily generalized to deal with strict and non-strict bounds. Such implementation considerations do not affect the results of this paper, and are thus not discussed any further.

plete is a ***partial trace*** over $\mathcal{A}$.

❑

Definition 4 [***Trace Satisfying a Constraint Set***] Let Ω be a well-behaved trace over an action set $\mathcal{A}$, C be a timing constraint set (i.e., a constraint relation) over $\mathcal{A}$, and Σ be the substitution $\{t(a_{ij}) := \tau_i,\ i = 1,\ldots, \theta - 1,\ j = 1,\ldots, j_i\}$. Ω *satisfies* C if:

- In the case where Ω is a *complete* trace: the conjunction of the propositions associated with the constraints of C (Definition 1) is true under the substitution Σ.
- Else (Ω is an *incomplete* trace): there exists a substitution Σ' for the time stamp variables of actions *not* present in Ω such that the conjunction of the propositions associated with the constraints of C is true under the substitution $\Sigma \cup \Sigma'$.

❑

Definition 5 [***Trace*** and ***trace set*** of an ***Action Diagram***] A trace Ω of an action diagram $AD = (S, \mathcal{A}, o, C)$ is a *complete trace* (Definition 3) over the action set $\mathcal{A}$, such that Ω *satisfies* (Definition 4) C. The ***trace set*** of AD, denoted *TraceSet(AD)*, is the set of all traces of AD.

❑

Definition 6 [***Trace Form***] Let A be a vector of n distinct actions, $A = (a_1,\ldots, a_n)$ and U be a vector of real numbers, $U = (\tau_1,\ldots, \tau_n)$, such that $t(a_i) = \tau_i$, $i = 1,\ldots, n$. *TraceForm*(U) is the well-behaved trace (Definition 3) over the set of actions $\{a_1,\ldots, a_n\}$ obtained by first partitioning the set $\{(a_i, \tau_i) \mid i = 1,\ldots, n\}$ into sets of (a_i, τ_i) pairs that have the same τ_i values, and then building a sequence of these sets in strictly increasing τ_i values.

❑

Definition 7 [***Constraint Graph***] Let $\mathcal{A}$ be an action set, C a set of constraints over $\mathcal{A}$ and, for any given pair of actions a_i, a_j of $\mathcal{A}$, let $C_{ij} = \{c_{ijk} \in C \mid \exists\ \delta_{ijk}, \exists\ \Delta_{ijk}, c_{ijk} = (a_i, a_j, \pi_{ijk})\}$, where the lower and upper bounds of π_{ijk} are δ_{ijk} and Δ_{ijk}, respectively. The *constraint graph* over $\mathcal{A}$ and C, denoted $CG(\mathcal{A}, C)$, or

simply *CG* (when $\mathcal{A}$ and C are clear from the context), is the directed weighted graph defined as follows:

- the vertex set of *CG* is $\mathcal{A}$
- for each pair of actions a_i, a_j, such that $a_i \neq a_j$, define w_{ij} as:

$$w_{ij} = Min\left(\underset{c_{ijk} \in C_{ij}}{Min}(\Delta_{ijk}) \ , \ \underset{c_{jik} \in C_{ji}}{Min}(-\delta_{jik}) \right)$$

  where the *Min* operator over an empty set is defined to yield infinity.
- the edge $e_{ij} = (a_i, a_j)$ exists and is of weight w_{ij}, iff w_{ij} is finite.

The set $C_{ij} \cup C_{ji}$ is the set of ***constraints associated with edge*** e_{ij}. We write *associated-constraints*$(e_{ij}) = C_{ij} \cup C_{ji}$.

❑

Note that the same set of constraints $C_{ij} \cup C_{ji}$ is associated with both edges, e_{ij} and e_{ji}. Note also that the above graph representation stems from the representation of the pair of inequalities $\delta_{ijk} \leq t(a_j) - t(a_i) \leq \Delta_{ijk}$ into the normalized form:

$$t(a_j) - t(a_i) \leq \ \Delta_{ijk}$$
$$t(a_i) - t(a_j) \leq -\delta_{ijk}$$

Let *CG* be a constraint graph over an action set $\mathcal{A}$, $a_i \in \mathcal{A}$, $a_j \in \mathcal{A}$. Given an edge $e_{ij} = (a_i, a_j)$, ***source***(e_{ij}) and ***sink***(e_{ij}) designate a_i and a_j, respectively. A ***path*** r is a sequence of edges $r = [e_1, \ldots, e_n]$, $n \geq 1$, such that *source*(e_i) = *sink*(e_{i-1}), for $i = 1, \ldots, n$. We say that the path is "from source(e_1) to sink(e_n)". The notations ***first(r)*** and ***last(r)*** refer to e_1 and e_n, respectively. The ***weight*** of a path r, denoted ***weight(r)*** is the sum of the weights of the edges of r. Note that as a consequence of Definition 7, the weight of any path of *CG* is finite.

Definition 8 [***Weak consistency***] Let AD = (S, $\mathcal{A}$, o, C) be an action diagram. Then, if there exists a complete trace over $\mathcal{A}$ that satisfies C, we say that C, *CG*($\mathcal{A}$, C), and AD are weakly consistent.

❑

Definition 9 [***dist***(a_i, a_j)] Let $\mathcal{A}$ be an action set, C a constraint relation over $\mathcal{A}$, and a_i, a_j, a pair of actions of $\mathcal{A}$, such that $a_i \neq a_j$. The ***maximum distance*** from a_i to a_j, denoted ***dist***(a_i, a_j), is defined as the maximum value of $t(a_j) - t(a_i)$ for

which there exists a complete trace over $\mathcal{A}$ that satisfies C.

❑

It can be shown [Tarj83] that $dist(a_i, a_j)$ is equal to the weight of the shortest (a_i, a_j) path in the graph $CG(\mathcal{A}, C)$. If C is weakly consistent, then it can be shown that the interval π_{ij} of lower and upper bounds $-dist(a_j, a_i)$, and $dist(a_i, a_j)$, respectively, is non-empty. Let d_{ij} be a real number such that there exists a complete trace Ω with $t(a_j) - t(a_i) = d_{ij}$ in Ω, and such that Ω satisfies C. Then, π_{ij} defines the unique largest set of real numbers $\{d_{ij}\}$. Applying Floyd's classical all-pairs shortest path algorithm [Tarj83] to the constraint graph $CG(\mathcal{A}, C)$ allows to determine the quantities $dist(a_i, a_j)$, for all (a_i, a_j) pairs and whether C is weakly consistent[1]. The algorithm is of $O(n^3)$ time complexity and $O(n^2)$ space complexity.

We will use the following terminology and notation: A path r from a_i to a_j in $CG(\mathcal{A}, C)$ is a ***tight path*** if its weight equals $dist(a_i, a_j)$. C is a ***tight constraint relation*** if, for all constraints (a_i, a_j, π_{ij}) of C, the lower and upper bounds of π_{ij} are equal to $-dist(a_j, a_i)$ and $dist(a_i, a_j)$, respectively. Given r_{ij}^1 and r_{ij}^2, two paths from a_i to a_j, we say that r_{ij}^1 is ***tighter than*** r_{ij}^2 if: $weight(r_{ij}^1) < weight(r_{ij}^2)$. We will use the notation $dist_{[CG]}(a_i, a_j)$ to emphasize the constraint graph (or sub-graph) over which $dist(a_i, a_j)$ is computed. Similarly, given an action diagram $AD = (S, \mathcal{A}, o, C)$, $dist_{[AD]}(a_i, a_j)$ indicates that $dist(a_i, a_j)$ is computed over the constraint graph defined by AD.

Definition 10 [***Port Soundness***] A weakly consistent action diagram $AD = (S, \mathcal{A}, o, C)$ is *port sound* if, for any two consecutive actions a_i^j, a_i^{j+1}, in the action sequence of every port p_i of AD, the relation $dist_{[AD]}(a_i^{j+1}, a_i^j) < 0$ holds.

Definition 11 [***Consistency***] An action diagram $AD = (S, \mathcal{A}, o, C)$ is *consistent* if it is weakly consistent and port sound. We say that C, $CG(\mathcal{A}, C)$, and AD are consistent, and we write *consistent*(C), *consistent*(CG), and *consistent*(AD).

❑

1. If the algorithm finds a negative $dist(a_i, a_i)$, for some a_i, i.e., a cycle of negative weight in CG, then C is inconsistent. Otherwise, it is weakly consistent.

3 Problems

An action diagram specification can be checked alone for consistency, which is a minimal form of realizability. Consistency checking allows to determine whether an occurrence time can be assigned to every action such that all constraints are satisfied and the specified order of occurrence of actions on a port is preserved. Another problem is the verification of the interface compatibility of communicating devices. In [Brzo91], this problem is addressed by checking that for each pair of actions related by an assume constraint c_a, the time distance between the same pair of actions as implied by the commit constraints is tighter than c_a. The notions of consistency and compatibility of action diagrams are insufficient for either constructing correct implementations or for verifying that two or more implementations will interact correctly when built according to their local specifications. We now illustrate these weaknesses.

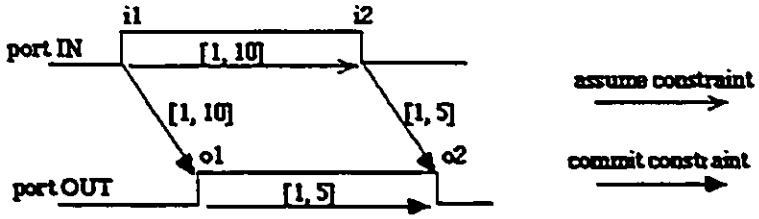


Figure 1:A non-causal specification.

3.1 Consistency

Consider the action diagram shown in Figure 1. $C = \{i1 \rightarrow i2\}_{assume} \cup \{i1 \rightarrow o1, o1 \rightarrow o2, i2 \rightarrow o2\}_{commit}$; the constraint system C is consistent and tight. When implementing a device according to this specification, the delay value for action o1 after the occurrence of action i1 has to be chosen from within the interval [1, 10]. However, this delay value depends on the selected occurrence time of the *in* action i2 which may occur after o1. For instance, if we choose $t_{o1} - t_{i1} = 1$ in the implementation, then if i2 occurs such that $t_{i2} - t_{i1} \in (5, 10]$ (which is within the specified limits) then there is no feasible occurrence time for o2. The environment would have to track the occurrence time of o1 and pro-

duce i2 after o1. Symmetrically, the implementation of the device could decide to do the same, await i2 and then produce o1, leading to a deadlock. Clearly, such a specification is non-causal as the decisions made by the device implementation depend on future actions of the environment, and vice versa. A possible solution is that the designer of the environment and the designer of the device analyze the action diagram and then agree on a joint strategy. Their decision is not part of the specification, however, hence it is impossible to implement each device independently and to verify compatibility of two devices strictly based on the action diagram specifications. It thus follows that consistency and tightness of C are not sufficient to guarantee a realizable specification, we must also consider *causality*. This situation is similar to the problem of non-realizability of ideal filters (with square frequency response) where the output of the filter would have to start changing before the arrival of a change on its input.

3.2 Compatibility

In [Brzo91], the authors propose verifying that the assume constraint values of one device are less tight than the time distances of the same actions produced (committed) by the other device. However, the method is exact only if each action diagram has ports and actions of only one direction (i.e., one action diagram has *in* actions, and the other one has *out* actions only). Otherwise, it can yield a false negative answer to the compatibility check.

Consider the two action diagrams in Figure 2. AD_1 indicates a simple delay from an *in* action on port p_1 to an *out* action on port p_2, while AD_2 drives p_1 depending on the *in* action i_3 on port p_2. Both specifications are realizable and devices built according to them can interact without violating the assumptions of their partners. Yet, the procedure of [Brzo91] will declare that the two action diagrams do not satisfy each other: the time distance between o_3 and o_4 in AD_1, as implied by the commit constraints of AD_1 is potentially ∞, while AD_2 assumes that this distance is in the interval [4, 10]. However, when the devices are put in communication (by connecting together same numbered ports), the time distance between i_3 and i_4 will fall within the assumed interval, because the time distance between actions o_1 and o_2 in AD_1 is dictated by the behavior of AD_2 (i.e., the commit of [3,3] from i_3 to o_2). This discrepancy arises because

the compatibility checking procedure of [Brzo91] does not take into account the composed behavior of the interconnected system.

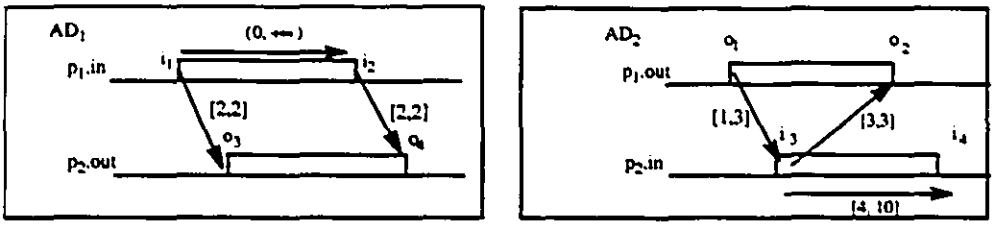
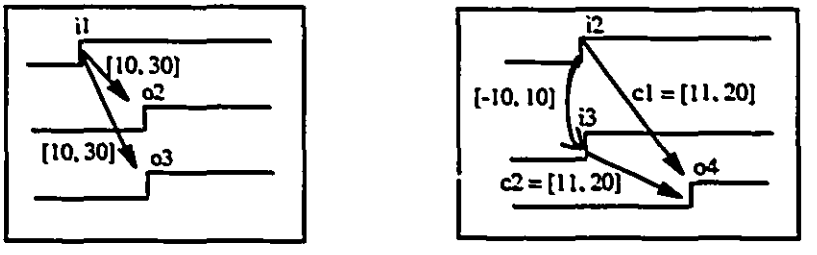


Figure 2: Assumed [4, 10] between i_3 and i_4 in AD_2 does not cover $(0, +\infty)$ between o_3 and o_4 produced in AD_1.



(a) AD_1 (b) AD_2

Figure 3: A simple composition of commit constraints does not work here.

A simple attempt to correct the compatibility checking procedure can yield **false positive** answers to the compatibility check. For example, we could compose the commit constraints of the two systems and verify that the resulting time distances between actions satisfy the assumptions made by each of the systems. This is illustrated in Figure 3. In AD_1, the *out* actions 2 and 3 can follow the *in* action 1 within [10, 30]. If an implementation is made according to this specification, it should be able to freely choose output delays in the specified intervals, for example, $t(o_2) - t(i_1) = 10$ and $t(o_3) - t(i_1) = 30$. In AD_2, the *out* action 4 is to be produced within the interval [11, 20] from both of the *in* actions 2 and 3, assuming that these actions occur within 10 units of time from each other. Both constraint systems are consistent and tight. If we now combine

the commit constraints of AD_1 and AD_2 to obtain the total system behavior, and then compute the distance between actions 2 and 3, we find that the assumption t(3) - t(2) ∈ [-10, 10] is satisfied. Yet, the implementation of AD_1 mentioned above would violate the assumptions made by AD_2 (and thus its implementation). This is because the convergent conjunctive commit constraints in AD_2 determine the position of actions 2 and 3 jointly with those of AD_1. That is, the positions of actions 2 and 3 in the implementation of AD_1 would have to be determined jointly with the occurrence time of the future action o4 produced by a different component of the system, clearly a non-causal task.

4 Block Machines

As implied by the preceding section, realizability of an action diagram specification depends not only on the consistency of the action diagram constraint system, but also on whether the action diagram describes a causal system. We propose the following intuitive description of a causal action diagram: The decision that an *out* (*in*) action a_i should occur at time $t(a_i)$ according to the action diagram commit (assume) constraints must not depend on the occurrence instants of actions that could be performed by the environment (device) at time $t \geq t(a_i)$. We do not eliminate the possibility that the occurrence time of an *out* action depends on future *out* action times (provided that they themselves do not depend on future *in* actions) and any past action times. This suggests that, in a causal action diagram, we should be able to partition the set of actions into ***blocks*** such that, within a block, local action time computations are possible depending only on past actions in preceding blocks. If such a partition exists, then the action diagram has a causal interpretation in the above sense and is considered as realizable. An action diagram together with some specified partition of its action set defines a machine, which we designate as ***block machine*** (BM). In this section, we formalize the structure and operational semantics of block machines, and we prove some basic properties of these machines that will be useful in developing the causality and compatibility criteria.

Definition 12 [*Block Machine*] A block machine (BM) is the quadruple $(\mathcal{A}, o, \mathcal{B}, \mathcal{T})$, where:

- $\mathcal{A}$ is a set of actions.
- o is the "origin" action, $o \in \mathcal{A}$. Let $\mathcal{A}' = \mathcal{A} - \{o\}$.

- $\mathcal{B}$ is a set of κ "blocks".
- $\mathcal{T}$, the "trigger relation", is a relation on $\mathcal{A} \times \mathcal{B}$. When a pair (a, B) is in $\mathcal{T}$ we say that "a is a trigger of B". The set of triggers of a block B is denoted *trigs(B)*.
- A block B_i of $\mathcal{B}$ is a pair (L_i, Φ_i), where :
  - $L_i \subseteq \mathcal{A}$. L_i is designated as the set of "local actions of B_i", or simply "actions of B_i". We will use the notation *actions(B_i)*. Given an action a of B_i, *block*(a) designates B_i.
  - $\Phi_i : \Re^{m_i} \rightarrow \wp(\Re^{n_i})$, i.e., Φ_i is a function from the set of real finite-valued vectors of dimension m_i to the set of sets of real finite-valued vectors of dimension n_i, where m_i is the number of triggers of B_i and n_i is the number of actions of B_i. The set returned by Φ_i, for any given input vector U_j can be empty, finite, or infinite. Φ_i is designated as the "time computation function" of block B_i.
- The set $\{L_i \mid i = 1,\ldots, \kappa\}$ is a partition over $\mathcal{A}$.

❑

Definition 13 [***Prec* Relation on $\mathcal{B}$**] Given a block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$), the binary relation *Prec* on $\mathcal{B}$ is: "B_i *Prec* B_j" if there is an action a_{ik} of B_i such that a_{ik} is a trigger of B_j. We say that B_i is a ***predecessor*** of B_j.

❑

Definition 14 [**"<" *Relation on* $\mathcal{B}$**] Given a block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$), the binary relation "<" on $\mathcal{B}$ is defined as follows. "$B_1 < B_m$" if there is a sequence[1] of blocks $[B_i, i = 1, \ldots, m]$ of $\mathcal{B}$ such that for each i, $i = 1, \ldots, m-1$, B_i *Prec* B_{i+1}.

❑

Obviously, "<" is a transitive relation. A trace of a block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$) is a trace over its action set $\mathcal{A}$ (Definition 3). Operationally, the trace is built by the procedure M_{exec} (Definition 16), given an arbitrary occurrence time t_0 for the origin action. An execution of this procedure is said to be an ***execution of M***.

---

1. Note: In this sequence, it does not matter whether $B_i \neq B_{i+1}$ or not.

Definition 15 [*Execution Model: Assumptions*] The following assumptions are made in defining the execution of a block machine:

- A time stamp variable $t(a_i) \in \Re$ is associated with each action a_i of $\mathcal{A}$; initially, for all a_i actions, $t(a_i) = \infty$.
- The predicate **occurs**(a_j, τ) is true iff $t(a_j) = \tau$. The action a_j is said to *occur* at time τ.
- There is a global time variable $T \in \Re$ that increases monotonically only when, and always when, the execution is *in a wait state*. Initially, the global time variable is reset to t_0 with the operator **reset**().
- The execution enters a wait state when the operator **wait**() is executed. This operator, applied to a set of actions, suspends the execution until the global time variable T reaches a value τ, such that $\exists j$, **occurs**(a_j, τ), where a_j is an action of the specified set. In *any* execution of the wait() operator, the global time increases by a *non-null* quantity.
- A boolean flag, **occurred**(a_i), is associated with each action a_i of $\mathcal{A}$; initially **occurred**(a_i) is false. The flag is set to true when the action occurs.
- The predicate **enabled**(B, τ) is true iff:
  $[\forall\ trig_j \in trigs(B)$, occurred($trig_j$, τ)$] \wedge [\exists\ trig_j \in trigs(B)$, occurs($trig_j$, τ)$]$.
  Block B is said to be *enabled* at time τ.
- $TRIG_k$ denotes the vector of trigger actions of block B_k.
- ACT_k denotes the vector of local actions of block B_k.
- $t(X_k)$, where X_k is a vector of actions, denotes the vector of time stamps of X_k.
- The operator **deadlock**() suspends the execution forever; if this operator is executed, the execution is said to *enter the deadlock state*.
- The function **choose**, applied to a non-empty set, returns a non-deterministically selected element of that set.
- The operator **update**(*Trace*, X_k) accepts a trace as its first argument and a vector X_k of actions as its second argument. The operator selects those actions x_{ki} of X_k that are such that $t(x_{ki}) > T$. Each such x_{ki} is inserted in the appropriate set ω_i of the trace according to $t(x_{ki})$, as per Definition 3.

□

Definition 16 [***Execution Model***] An execution of a block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$) is an execution of the procedure M_{exec} defined as follows:

```
Procedure M_exec(t_0)
begin
    t(o) := t_0 ;   reset( T, t_0^- ) ;   Trace := [{(o, t_0)}] ;
    while ∃ a_i ∈ A , not (occurred(a_i)) do
        wait(A) ;
        for all i such that occurs(a_i , T) do
            occurred(a_i) := true;
        end for;
        for all k, such that enabled(B_k , T) do
            if Φ_k(t(TRIG_k)) = ∅
             then deadlock() ;
             else
                t(ACT_k) := choose(Φ_k(t(TRIG_k))) ;
                update(Trace, ACT_k) ;
            end if;
        end for;
    end while;
    success;
end procedure.
```

We use the following terminology. The ***execution*** of a block B_j consists of executing the iteration $k = j$ of the loop "for all k, such that enabled(B_k, T) ... end for". When this iteration is completed (either by executing the statement deadlock(), or the statement update(*Trace*, ACT_j)), we say that block B_j has *executed*. The ***execution of M up to a block*** B_j is the execution of the procedure M_{exec} until and including the execution of block B_j.

❑

Note that, by definition, there is a single trace associated with any given execution. However, due to the parameter t_0 and the choices made by the ***choose*** function in M_{exec}, there is a set of executions, denoted by *executions*(M), and hence a *set* of traces associated with a block machine. The semantics of a block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$) are given by its trace set.

Definition 17 [***Trace set of a block machine***] Given a block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$), the trace set of *M*, denoted *TraceSet*(*M*), is the set of *complete* traces (Definition 3) over $\mathcal{A}$ that are generated by all possible executions of M, for all possible values of the parameter t_0. ❑

Lemma 1 [*Non-Zeno Time*] Let B be an arbitrary block of $\mathcal{B}$ in a block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$). For B to become enabled at some time t, it must be that for all i, such that $B_i' < B$, B_i' was enabled at some time $t_i' < t$.

Proof. From Definition 16, the definition of the **enabled** predicate, and the assumption of monotonically increasing time (in Definition 15), it follows that for B to become enabled at some time t, all the triggers of B must have occurred at times smaller than, or equal to t. From Definition 16, there is at least one execution of the wait() operator between the setting of the time stamp of an action, and the occurrence of that action. From the assumptions in Definition 15, there is a non-null amount of time that passes in any execution of the wait() operator. Hence, the blocks containing the triggers of B, i.e., all B_i' blocks such that B_i' $\mathcal{P}rec$ B, must have been enabled at times *strictly* smaller than t. Carrying this argument inductively over "chains" of consecutive pairs of blocks related by the $\mathcal{P}rec$ relation, we obtain that all blocks B_i', such that $B_i' < B$, must have been enabled at times *strictly* smaller than t.

❑

Lemma 2 [*"At Most Once" Action Occurrence*] In any execution of a block machine M, where M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$), all actions of $\mathcal{A}$ occur at most once.

Proof. Define the sets $\mathcal{B}_c$, $\mathcal{B}_1$, and $\mathcal{B}_2$ as follows:

$$\mathcal{B}_c = \{ B_i \in \mathcal{B} \mid B_i < B_i \}$$
$$\mathcal{B}_1 = \{ B_i \in \mathcal{B} \mid \neg(B_i < B_i) \wedge \neg\exists B_j \in \mathcal{B}_c, B_j < B_i \}$$
$$\mathcal{B}_2 = \{ B_i \in \mathcal{B} \mid \neg(B_i < B_i) \wedge \exists B_j \in \mathcal{B}_c, B_j < B_i \}$$

Consider a block $B_c \in \mathcal{B}_c$. From Lemma 1, and due the fact that $B_c < B_c$ (which stems from $B_c \in \mathcal{B}_c$), it follows that in order for B_c to become enabled at some time t, it must be that B_c was enabled at some time $t' < t$. Carrying this argument recursively, we obtain that this is only possible if B_c was enabled at $T = t_0$. From Definition 16, the only blocks that can be enabled at $T = t_0$ are those that have the origin action o as their only trigger. However, a block B that has o as its only trigger, has no predecessor blocks, i.e., there is no B_i' such that B_i' $\mathcal{P}rec$ B, and hence no B_i' block such that $B_i' < B$. This contradicts the assumption that $B_c < B_c$. Therefore, B_c is never enabled, its actions never occur, and hence the lemma holds for all actions of any $B_c \in \mathcal{B}_c$.

Consider an arbitrary block B_2 of $\mathcal{B}_2$. From Lemma 1, and from the definition of $\mathcal{B}_2$, it follows that there exists at least one block B_j, $B_j \in \mathcal{B}_c$, such that the enabling of B_j is a prerequisite for the enabling of B_2. Since the blocks of $\mathcal{B}_c$ are never enabled, it follows that B_2 is never enabled. Thus its actions never occur and the lemma holds for all actions of any $B_2 \in \mathcal{B}_2$.

The only blocks that can become enabled during an execution, are the blocks of $\mathcal{B}_1$. In the *WHILE* loop of Definition 16, consider those iterations in which there is at least one enabled block; designate these iterations as "*enabling iterations*". If there are no enabling iterations, then no block of $\mathcal{B}_1$ is ever enabled, and hence the lemma is true for all actions of all blocks of M. If, on the other hand, there are enabling iterations, then assign consecutive integers i, $i \geq 1$,to consecutive enabling iterations. Let $E(i)$ be the set of blocks enabled at enabling iteration i, $i \geq 1$, $PE(i)$ be the set of blocks enabled at some enabling iteration j, where $1 \leq j < i$, with $PE(1) = \varnothing$, and $t(i)$ designate the value of the global time variable, T, at iteration i. Define the property $S(i)$ as $S(i) = [E(i) \cap PE(i) = \varnothing]$, i.e., $S(i) = True$ means that the blocks enabled at enabling iteration i have never been enabled before. In the following, we prove by induction over enabling iterations i, that $S(i)$ holds for all i, and thus that all blocks of $\mathcal{B}$ are enabled at most once.

- *Induction base* ($i = 1$): $PE(1) = \varnothing$, hence $E(1) \cap PE(1) = \varnothing$, and thus $S(1) =$ true.

Before going to the induction step, we note that, from Definition 16, the first iteration of the *WHILE* loop is at $T = t_0$. If there are blocks enabled at this iteration, then this iteration is also the first *enabling* iteration, and hence $t(1) = t_0$. On the other hand, if there are no blocks enabled at the first iteration (i.e., at $T = t_0$), then, in the course of this iteration, no action of $\mathcal{A}$ will have its time stamp set, and hence the execution will be suspended forever when the wait() operator is executed at the second iteration. As a result, no action of $\mathcal{A} = \mathcal{A} - \{o\}$ ever occurs in this execution. The only action to occur is the origin action o which occurs only once at $T = t_0$. Hence the lemma is true for all actions of $\mathcal{B}$. The following induction step is thus relevant only in the case where the first *enabling* iteration ($i = 1$) is such that $t(1) = t_0$.

- *Induction step:* The induction hypothesis is $[S(1) \wedge \ldots \wedge S(i)]$. We want to prove $S(i+1)$. Consider a block B_k enabled at enabling iteration $i+1$, i.e., $B_k \in E(i+1)$. From the definition of the *enabled* predicate (see Definition 15), it follows that all actions of $TRIG_k$ (the vector of triggers of B_k) have occurred in the time interval $[t(1), t(i+1)]$. Consider a given trigger, $trig_{kn}$, of $TRIG_k$. If $trig_{kn} \in \mathcal{A}$, then from Definition 16, the occurrence of $trig_{kn}$ in the time interval $[t(1), t(i+1)]$ implies that $block(trig_{kn})$ was enabled at some enabling iteration j, i.e., $block(trig_{kn}) \in E(j)$, for some j, where $1 \leq j \leq i$. From the induction hypothesis, we have $[S(1) \wedge \ldots \wedge S(i)]$, i.e., any block enabled anywhere in the enabling iteration interval $[1, i]$ is enabled *once* in that interval. This in turn implies that $trig_{kn}$ has occurred only once in the time interval $[t(1), t(i+1)]$. If $trig_{kn} \notin \mathcal{A}$ (i.e., $trig_{kn} = 0$), then from Definition 16, $trig_{kn}$ has occurred only once at $t(1)$. Thus all triggers of B_k have occurred only once in the time interval $[t(1), t(i+1)]$.

  In addition, from the definition of the *enabled* predicate (Definition 15), the fact that B_k is enabled at the enabling iteration $i+1$ implies that there is at least one trigger of B_k that occurs *exactly* at time $t(i+1)$. Let $trig_{kn}$ be such a trigger. Since in the previous paragraph we have shown that all actions of $TRIG_k$ occur *once* in the interval $[t(1), t(i+1)]$, it follows that $trig_{kn}$ does *not* occur anywhere in $[t(1), t(i)]$. Hence B_k is not enabled anywhere in the enabling iteration interval $[1, i]$, i.e., $B_k \notin PE(i+1)$. Since, by assumption, B_k is an arbitrary block such that $B_k \in E(i+1)$, and since we have just proven that $B_k \notin PE(i+1)$, it follows that $E(i+1) \cap PE(i+1) = \emptyset$. Hence, $S(i+1) =$ true.

It follows that all blocks of $\mathcal{B}$ are enabled at most once in an execution. From Definition 16, this implies that the time stamp of each action is set at most once in the execution. Due to the assumption of monotonically increasing global time and the definition of the *occurs* predicate (both in Definition 15), it follows that each action occurs at most once in an execution.

□

Lemma 3 [*Trace Well-Behavedness*] The trace associated with an execution of a block machine $M = (\mathcal{A}, 0, \mathcal{B}, \mathcal{T})$ is well-behaved.

Proof. In the course of the proof of Lemma 2, we have shown that all blocks of $\mathcal{B}$ are enabled at most once in an execution. From Definition 16, the only time that the trace is possibly updated with the actions of a block, is when this block is enabled. Hence, each action of $\mathcal{A}$ can be present at most once in the execution trace, and thus the trace is well-behaved.

❑

Definition 18 [***Execution Termination***] An execution E of a block machine is said to *terminate* if the "*success*" statement of Definition 16 executes in E.

❑

Lemma 4 [***Trace Completion and Execution Termination***] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$) be a block machine, E an execution of M, and Ω the trace produced by E. Then, Ω is complete iff E terminates.

Proof (*if*). Assume that an execution terminates. Due to the *WHILE* loop condition of Definition 16, this implies that, for each $a_i \in \mathcal{A}$, occurred(a_i) was set to true at $T = \tau_i$, for some finite τ_i, which implies that occurs(a_i, τ_i) was true at $T = \tau_i$. From the definition of the *occurs()* predicate, this implies that the time stamp, $t(a_i)$, of a_i was set. This in turn implies that the trace *update()* operator (Definition 15) was applied to a_i , since from Definition 16 this operator is applied only when action time stamps are set. In addition, since T is monotonically increasing, the only way that occurs(a_i, τ_i) could have been true at $T = \tau_i$ is that $t(a_i)$ was set to τ_i when T was equal to some $\tau_i' < \tau_i$. This is exactly the condition under which the trace *update()* operator inserts a_i in the trace. It follows that a_i , and hence each action of $\mathcal{A}$, is inserted in the trace at least once. Since, from Lemma 3, each action is present at most once in the trace, it follows that each action of $\mathcal{A}$ appears *exactly* once in the trace, and hence the trace is complete.

Proof (only if). Consider an arbitrary execution that yields a complete trace. Since the trace is complete, it must be that the *update*() operator has updated the trace with all actions. Hence, it must be that the execution has invoked the *update*() operator on all actions. Since, from Definition 16, the *update*() operator is invoked on the actions of a block B_k only when the following two conditions are met: 1- B_k is enabled and 2- $\Phi_k(t(TRIG_k)) \neq \varnothing$, it follows that a complete trace necessarily implies that in the time interval starting at $T = t_0$ to the time at which the trace becomes complete, all blocks are enabled, and the

first time that any given block B_k becomes enabled, the set $\Phi_k(t(TRIG_k))$ is not empty. Now, since from Lemma 2, each block is enabled at most once in any execution, it follows that all blocks are enabled exactly once, and hence at no enabling of any block B_k does the corresponding $\Phi_k(t(TRIG_k))$ yield the empty set. Hence the deadlock state is never entered.

In addition, since: 1- T (the global time) is monotonically strictly increasing, 2- (from Definition 15) the *update*() operator updates the trace with an action a_j at time $T = T_i$, only if $t(a_j)$ was set to a value such that $t(a_j) > T_i$ and 3- $t(a_j)$ is finite (due to the assumption on the Φ's in Definition 12), it follows that the execution cannot get suspended forever in a *WAIT* state. Since the execution never enters the deadlock state, nor suspends forever in a *WAIT* state, it follows that any action that has had its time stamp set, will occur. Since all blocks are enabled exactly once (as we have shown above), and since upon a block enabling, the time stamp of all actions of the block are set, it follows that all actions have their time stamp set, and hence all actions occur.

Consider the last *WHILE* loop iteration at which there is an action that occurs, and designate this iteration as I_f. After the *occurred* flags of actions are updated in this iteration, the proposition $[\forall a_i \in \mathcal{A}, occurred(a_i)]$ becomes true. In addition, it must be that all blocks have already been enabled before the I_f iteration is entered (or else, not all actions could have occurred). Since, from Lemma 2, we know that all blocks are enabled at most once in any execution, it follows that in the I_f iteration there will be no enabled blocks. Hence, the loop "for all k, such that $enabled(B_k, T)$ do ... end for" in Definition 16, will not be entered, and thus the I_f iteration will immediately terminate. There will not be a subsequent iteration of the *WHILE* loop, as the loop predicate "$\exists\, a_i \in \mathcal{A}$, not($occurred(a_i)$)" will yield false. Hence, the next statement to execute is "*success*".

□

Definition 19 [***Live Block Machine***] A block machine M is live if, for every execution E_i of M, there exists a finite value τ_i, such that the trace associated with E_i is *complete* (Definition 3) , at $T = \tau_i$ (where T is the global time variable).

□

Definition 20 [***Forward Time Property***] Consider a block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$). Let E_i be an execution of M. Let U_j^i denote $t(TRIG_j)$ in E_i, where $TRIG_j$ is the vector of trigger actions of block B_j. Let Φ_j be the time computation function of B_j and V_j^i the time vector chosen by the execution E_i from the set $\Phi_j(U_j^i)$. Let m_j and n_j be the number of triggers and actions, respectively, of block B_j. We write U_j^i and V_j^i in the form $(u_{j1}^i, \ldots, u_{jm_j}^i)$ and $(v_{j1}^i, \ldots, v_{jn_j}^i)$, respectively. Then, we say that:

- A ***block B_j satisfies the forward time property in execution E_i*** if B_j is enabled in E_i at some finite τ and the property $v_{jl}^i > u_{jh}^i$, $h = 1, \ldots, m_j$, $l = 1, \ldots, n_j$, holds.
- The ***execution E_i up to a block B_j satisfies the forward time property*** if all blocks executed up to B_j in E_i satisfy the forward time property in E_i. The ***execution E_i satisfies the forward time property*** if all the blocks of M satisfy the forward time property in E_i.
- The block ***machine*** M ***satisfies the forward time property*** if all its executions satisfy the forward time property.

□

Lemma 5 [***Equivalent Liveness***] Given a block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$),

[M is live] $\Leftrightarrow$ [

M satisfies the ***forward time*** property	(P1)
$\wedge$ each block of $\mathcal{B}$ has ***at least one trigger***	(P2)
$\wedge$ the "$<$" relation on $\mathcal{B}$ is a ***partial order***[1]	(P3)
$\wedge$ no execution of M enters the ***deadlock*** state	(P4)

]

Proof ($\Leftarrow$). The proof is by contradiction. Assume that statements P1 to P4 hold and that M is not live. By Definition 19, this implies that there exists a trace of M that does not reach completion, and hence, by Lemma 4, an execution which never terminates. From Definition 16, it is clear that an execution that does not terminate must be forever suspended either in the deadlock state, or in a wait state. The former situation contradicts P4. The latter

1. A partial order is a binary relation R such that R is transitive and, for every x in the field of R, $x\,R\,x$ is false.

case implies that there is at least one action that has not occurred and which never occurs. This in turn implies that:

1. Either there is an action for which the time stamp has been set to a time value that is not strictly greater than the current value of T. This contradicts P1.

2. Or the block in which this action is, is never enabled. However, for this to happen without the execution having already entered the deadlock state (the assumption is that the suspension is in a wait state), it must be that:

   2.1. Either there exists a block with no triggers. This contradicts P2.

   2.2. Or there is a cyclic dependency in the trigger relation, i.e., the "<" relation on $\mathcal{B}$ is not a partial order. This contradicts P3.

Hence in all situations, a contradiction is obtained.

Proof ($\Rightarrow$). The proof is again by contradiction. Assume that M is live and at least one of the statements P1 to P4 does not hold. This, however, trivially implies that there is at least one execution that does not terminate, which by Lemma 4 implies that there is a trace that does not reach completion, thus implying that M is not live – contradiction.

□

5 From Action Diagrams to Block Machines

To reason about the causality and compatibility of consistent action diagrams, we will map these onto block machines, designated as ***derived block machines***. The aim is to obtain a live block machine with the same trace set as that of the original action diagram. The mapping is uniquely defined, given a consistent action diagram and a partition over its action set.

The time computation functions of a derived block machine are described by the "local constraint" sets $\{C_i \mid i = 1,\ldots, \kappa\}$. Such a machine can be seen as a definitional refinement (or special case) of the block machine in Definition 12. To emphasize this refinement, we extend the structural definition of a block machine to $M = (\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$, where C is the constraint set of the action dia-

gram. We also extend the structural definition of a block B_i to be the triplet (L_i, C_i, Φ_i).

Definition 21 [***Derived Block Machine***] Consider a consistent action diagram AD = (S, $\mathcal{A}$, o, C), where $C = C' \cup C^0$, and let $\mathcal{P}$ be a partition over $\mathcal{A}' = \mathcal{A} - \{o\}$, such that $\mathcal{P} = \{L_i \mid i = 1,\ldots, \kappa\}$. The block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) *derived* from the pair (AD, $\mathcal{P}$), denoted ***dBMs***(AD, $\mathcal{P}$), is such that:

- $\mathcal{B}$ is a set of blocks $\{B_i \mid B_i = (L_i, C_i, \Phi_i), i = 1,\ldots, \kappa\}$.
- $\mathcal{T} = \{(a_i, B_k) \mid a_i \in \mathcal{A} \wedge B_k \in \mathcal{B} \wedge (a_i = o \vee block(a_i) \neq B_k)$
  $\wedge \exists a_j \in actions(B_k), \exists \pi \in I,$
  $[(a_i, a_j, \pi) \in C \vee (a_j, a_i, \pi) \in C]$
  $\wedge [dist_{[AD]}(a_j, a_i) < 0 \vee (dist_{[AD]}(a_j, a_i) \geq 0 \wedge dist_{[AD]}(a_i, a_j) \geq 0)]\}$.
- C_i, the *local constraint relation* of block B_i, $i = 1,\ldots, \kappa$, is:
  $C_i = \{ c \in C \mid \exists a_j \in L_i, \exists a_k \in \mathcal{A}, \exists \pi \in I,$
  $[c = (a_j, a_k, \pi) \vee c = (a_k, a_j, \pi)]$
  $\wedge [a_k \in L_i \vee (a_k, B_i) \in \mathcal{T}]\}$.
- Φ_i, the *time computation function* of block B_i, is defined as:

  $$\Phi_i(X_i) = \{ V_i \mid TraceForm(concat(X_i, V_i)) \text{ satisfies } C_i \}$$

  where *TraceForm*() is the operator defined in Definition 6, the "*satisfies*" predicate is as per Definition 4, X_i is the vector of time stamp variables of the triggers of block B_i, V_i is some value of the vector of time stamp variables of the local actions of B_i, and $concat(X_i, V_i)$ indicates the vector which components are the concatenation of the components of the vectors X_i and V_i. Note that, due to the linear form of the constraints in C_i, $\Phi_i(X_i)$, for a given value of X_i, describes a polyhedron in the space of dimension n_i, where n_i is the number of actions of block B_i.

□

Notation : ***dBMs***(AD) denotes the set of block machines derived from AD, i.e., M $\in$ ***dBMs***(AD) if and only if there exists a partition $\mathcal{P}$ of the action set of AD such that M = ***dBM***(AD, $\mathcal{P}$).

6 Formalizing the Concept of Causality

Definition 22 [*WDT(M)*] Let M be a block machine derived from a consistent action diagram, where M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C), and let $CG = CG(\mathcal{A}, C)$. M is said to satisfy the *well-defined triggers* property, denoted *WDT(M)*, if:

$\forall B_i \in \mathcal{B}$, $\forall trig_{ij} \in trigs(B_i)$, $\forall a_{ik} \in actions(B_i)$, $dist_{[CG]}(a_{ik}, trig_{ij}) < 0$.

❑

Definition 23 [*Local Path*] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) be a derived block machine (Definition 21) and consider a set Q of blocks, $Q \subseteq \mathcal{B}$. An edge e of $CG(\mathcal{A}, C)$ is *local* to Q if there exists a block $B_j = (L_j, C_j, \Phi_j)$ of Q such that *associated-constraints*$(e) \subseteq C_j$. An edge e is *local* to a block B_i if e is local to $\{B_i\}$. A path r of $CG(\mathcal{A}, C)$ is *local* to Q (B_i) if all the edges of r are local to $Q(B_i)$.

❑

Definition 24 [*Past*(a_j), *Past*(a_i, a_j)] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$) be a block machine such that "$<$" is a partial order on $\mathcal{B}$, and let $a_i \in \mathcal{A} - \{o\}$, $a_j \in \mathcal{A}$, and $a_k \in \mathcal{A}$. We define *past*(a_j) and *past*(a_j, a_k) as follows:

- $past(o) = \emptyset$
- $past(a_i) = \{B_l \in \mathcal{B} \mid B_l < block(a_i) \vee B_l = block(a_i)\}$
- $past(a_j, a_k) = past(a_j) \cup past(a_k)$.

❑

Definition 25 [*Past-dominated(M)*] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) be a block machine derived from a consistent action diagram AD, such that M satisfies the well-defined triggers property (Definition 22). M is said to satisfy the *past-dominated* property, denoted *past-dominated(M)*, if:

$\forall a_i \in \mathcal{A}$, $\forall a_j \in actions(\mathcal{A}) - \{a_i\}$,

$\forall q_{ijk}$ tight path from a_i to a_j in $CG(\mathcal{A}, C)$, q_{ijk} is local to *Past*(a_i, a_j) .

❑

We propose the following formalization of causality (Definition 26 and Definition 27).

Definition 26 [***Causal Derived Block Machine***] A ***derived*** block machine M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) with $\mathcal{B} = \{(L_i, C_i, \Phi_i) \mid i = 1,\ldots, \kappa\}$ is ***causal*** if the following three conditions hold:

1. $\forall i, i = 1,\ldots, n$:
   - all actions in L_i have the same direction (designated as the *block direction*), and
   - all constraints in C_i have an intent which is "compatible" with the block direction (i.e., *commit* intent for *out* blocks and *assume* intent for *in* blocks).
2. *WDT(M).*
3. *past-dominated(M).* ❑

Notation : ***CdBMs(***AD*)* denotes the set of *causal* block machines derived from AD.

Definition 27 [***Causal Action Diagram***] An action diagram AD = ($\mathcal{S}$, $\mathcal{A}$, o, C) is causal if there exists a partition $\mathcal{P}$ of its action set $\mathcal{A}$ such that the derived block machine (Definition 21), *dBM(*AD, $\mathcal{P}$*)*, exists and is causal (Definition 26). ❑

In the next section, we give some basic results regarding the solution space of linear constraint systems of action diagrams (Definition 2). These results will be useful in proving sufficient conditions for the liveness of a block machine (Section 8).

7 Time Zones

Definition 28 [***Time Zone***] Consider $\mathcal{A} = \{a_1,\ldots, a_n\}$ a set of n actions, $n \geq 1$, and C a set of timing constraints over $\mathcal{A}$ (Definition 2). The *time zone* (or simply *zone*), *Zone*($\mathcal{A}$, C), is the set of n-dimensional vectors $\{V_i = (t(a_1), \ldots, t(a_n)) \mid a_i \in \mathcal{A} \wedge V_i$ satisfies $C\}$. Due to the form of the constraints in C, Z is a polyhedron. ❑

Definition 29 [*Zone and Vector Projection*] Given a zone $Z = Zone(\mathcal{A}, C)$, a vector $V \in Z$, and $\mathcal{A}_s \subseteq \mathcal{A}$ the zone $Z_s = Z\Downarrow_{\mathcal{A}_s}$ (the vector $V_s = V\Downarrow_{\mathcal{A}_s}$) is the *projection* of Z (of V) onto the space of time vectors of $\mathcal{A}_s$.

❑

Definition 30 [*Product of Zones*] Given two zones $Z_1 = Zone(\mathcal{A}_1, C_1)$ and $Z_2 = Zone(\mathcal{A}_2, C_2)$, the product zone $Z_1 \otimes Z_2$ is $Zone(\mathcal{A}_1 \cup \mathcal{A}_2, C_1 \cup C_2)$.

❑

The following lemma holds due to the relative nature of the timing constraints of Definition 2 (i.e., bounds on time differences between action pairs). This lemma, as well as the two lemmas that immediately follow it, are well-known results [Dill89] and are given here without proof.

Lemma 6 [*Zone Relativity*] Consider a non-empty zone $Z = Zone(\mathcal{A}, C)$ with $\mathcal{A} = \{a_1, \ldots, a_n\}$. Then, $Z\Downarrow_{\{a_i\}} = \Re$, $1 \le i \le n$, i.e., the projection of Z onto any subspace of dimension 1 yields the complete real axis.

❑

Lemma 6 implies that if the occurrence time of any single action a_i, $1 \le i \le n$, is arbitrarily fixed to a real value t, then there exists a vector V of Z, such that the *ith* component of V is t. The next lemma addresses the canonical representation of time zones of dimension *strictly* greater than 1.

Lemma 7 [*Canonical Time Zone Representation*] A non-empty zone $Z = Zone(\mathcal{A}, C)$ of dimension $n \ge 2$, can be represented in a finite and canonical manner by an $n \times n$ matrix M, $M = [m_{ij}]$, $i = 1, \ldots, n$, $j = 1, \ldots, n$, with $m_{ij} = dist_{[Z]}(a_i, a_j)$. In addition, referring to Definition 29, the canonical form of $Z\Downarrow_{\mathcal{A}_s}$, when $|\mathcal{A}_s| \ge 2$, is obtained from the canonical form of Z, by deleting from the latter the rows and columns corresponding to actions in $\mathcal{A} - \mathcal{A}_s$.

❑

Notation: Given a constraint graph $G = CG(\mathcal{A}, C)$ and the corresponding zone $Z = Zone(\mathcal{A}, C)$, the notations $dist_{[G]}(a_i, a_j)$ and $dist_{[Z]}(a_i, a_j)$ are used interchangeably.

Lemma 8 [***Zone Comparisons***] Given two zones $Z_1 = Zone(\mathcal{A}, C_1)$ and $Z_2 = Zone(\mathcal{A}, C_2)$, where Z_1 is non-empty:

$$Z_1 \subseteq Z_2 \Leftrightarrow [\ \forall a_i \in \mathcal{A}, \forall a_j \in \mathcal{A}, dist_{[Z_1]}(a_i, a_j) \leq dist_{[Z_2]}(a_i, a_j)\ ].$$
$$Z_1 = Z_2 \Leftrightarrow [\ \forall a_i \in \mathcal{A}, \forall a_j \in \mathcal{A}, dist_{[Z_1]}(a_i, a_j) = dist_{[Z_2]}(a_i, a_j)\ ].$$

❑

Terminology: When Z_1 is non-empty and $Z_1 \subseteq Z_2$, we say that Z_1 and C_1 satisfy C_2.

Lemma 9 [***Zone Coverage***] Consider $G_1 = CG(\mathcal{A}_1, C_1)$ and $G_2 = CG(\mathcal{A}_2, C_2)$, two constraint graphs such that $G_p = CG(\mathcal{A}_1 \cup \mathcal{A}_2, C_1 \cup C_2)$ is consistent. Let Z_1 and Z_2 designate the zones associated with G_1 and G_2, respectively, and let $\mathcal{A}_{12} = \mathcal{A}_1 \cap \mathcal{A}_2$. Then, if $|\mathcal{A}_{12}| \leq 1$ or if $Z_1 \Downarrow \mathcal{A}_{12} \subseteq Z_2 \Downarrow \mathcal{A}_{12}$, then $Z_p = Z_1 \otimes Z_2$ is such that $Z_p \Downarrow \mathcal{A}_1 = Z_1$, i.e., Z_p "covers" Z_1.

Proof. If $\mathcal{A}_1$ contains a single action a, then $dist_{[G_p]}(a, a)$ trivially equals $dist_{[G_1]}(a, a)$ - they are both equal to 0 - and hence $Z_p \Downarrow \mathcal{A}_1 = Z_1$. In the following, we assume that $|\mathcal{A}_1| > 1$. Let a_i and a_j be two actions of $\mathcal{A}_1$, such that $a_i \neq a_j$. Consider an acyclic path q from a_i to a_j in G_p. If $|\mathcal{A}_{12}| \leq 1$, then clearly q cannot contain edges of G_2. Hence, for any pair (a_i, a_j), the shortest path from a_i to a_j in G_p is a path in G_1, and thus $Z_p \Downarrow \mathcal{A}_1 = Z_1$. Otherwise ($|\mathcal{A}_{12}| > 1$), since q starts and ends on vertices of G_1, the only way that q can contain edges of G_2 is that, for each edge e of G_2 in q, there exist a_k and a_l actions of $\mathcal{A}_{12}$, $a_k \neq a_l$, and a subpath r of q, such that r is from a_k to a_l, r is made up of edges of G_2 only and e is an edge of r. More precisely, if P^1 designates the set of paths of G_1 and P^2_{12} designates the subset of paths of G_2 that start and end on actions of $\mathcal{A}_{12}$, then q can be written as the following regular path expression, wherein a set represents a choice over its elements, and "+", ".", and "*" indicate choice, concatenation, and Kleene closure, respectively:

$$q = P^1 . (P^1 + P^2_{12})^* . P^1$$

From the lemma premise, we have $Z_1 \Downarrow \mathcal{A}_{12} \subseteq Z_2 \Downarrow \mathcal{A}_{12}$, hence it follows that for any a_k, a_l pair of actions of $\mathcal{A}_{12}$, such that $a_k \neq a_l$, $dist_{[G_1]}(a_k, a_l) \leq dist_{[G_2]}(a_k, a_l)$. Thus, for any subpath P^2_{12} of q between a pair of actions of $\mathcal{A}_{12}$, there is a path in G_1 of smaller or equal weight between the same pair of actions. Hence, the shortest distance from a_i to a_j in G_p is determined by a path in G_1, and therefore $dist_{[G_p]}(a_i, a_j) = dist_{[G_1]}(a_i, a_j)$. Since this is true for

any (a_i, a_j) pair of actions of $\mathcal{A}_1$, $a_i \neq a_j$, it follows that $Z_{p \| \mathcal{A}_1} = Z_1$.

□

We will use the following terminology. Given an action diagram $AD = (\mathcal{S}, \mathcal{A}, o, C)$, and a derived block machine $M = (\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$, we will refer to $Zone(\mathcal{A}, C)$ as the ***global zone of AD*** and/or the ***global zone of M***.

8 Liveness of Derived Block Machines

Lemma 10 [***WDT(M)*** $\Rightarrow$ ***"<" is a partial order***] Let M = $(\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$ be a block machine derived from a consistent action diagram. Then, *WDT(M)* implies that the "<" relation on $\mathcal{B}$ is a partial order.

Proof. The proof is by contradiction. Assume $CG(\mathcal{A}, C)$ is consistent, *WDT(M)* holds, and "<" on $\mathcal{B}$ is not a partial order. The latter implies that there exists a block B_i such that $B_i < B_i$, which in turn implies that there exists a sequence of blocks [B_i, B_{i+1}, ..., B_{i+m}, B_i], with $m \geq 0$, such that $B_i \; \mathcal{P}rec \; B_{i+1}$, $B_{i+1} \; \mathcal{P}rec \; B_{i+2}$, ..., $B_{i+m-1} \; \mathcal{P}rec \; B_{i+m}$, and $B_{i+m} \; \mathcal{P}rec \; B_i$. From the *WDT(M)* property, this implies (where L_j is the set of actions of block B_j) that :

$$\exists\, a_{j_i} \in L_i, \forall\, a_{k_{i+1}} \in L_{i+1},\ dist(a_{k_{i+1}}, a_{j_i}) < 0$$
$$\exists\, a_{j_{i+1}} \in L_{i+1}, \forall\, a_{k_{i+2}} \in L_{i+2},\ dist(a_{k_{i+2}}, a_{j_{i+1}}) < 0$$
$$\ldots\ldots$$
$$\exists\, a_{j_{i+m-1}} \in L_{i+m-1}, \forall\, a_{k_{i+m}} \in L_{i+m},\ dist(a_{k_{i+m}}, a_{j_{i+m-1}}) < 0$$
$$\exists\, a_{j_{i+m}} \in L_{i+m}, \forall a_{k_i} \in L_i,\ dist(a_{k_i}, a_{j_{i+m}}) < 0.$$

By simple transitivity of the arithmetic "<", this implies that $\forall a_{k_i} \in L_i$, $dist(a_{k_i}, a_{j_i}) < 0$ and, since $a_{j_i} \in L_i$, we get in particular, that $dist(a_{j_i}, a_{j_i}) < 0$. However, this means that $CG(\mathcal{A}, C)$ is inconsistent – contradiction.

□

For the rest of this paper, we extend the *dist* notation (Definition 9) as follows: given a set Q of blocks, the notation $dist_{[Q]}(a_i, a_j)$ indicates the length of the shortest path from a_i to a_j in the graph in which the vertex set is composed of the union of the local action sets and trigger sets of the blocks of Q, and the edge set corresponds to the union of the local constraints of the blocks of Q. In addition, if B is a block, the notation $dist_{[B]}(a_i, a_j)$ is equivalent to $dist_{[\{B\}]}(a_i, a_j)$.

Lemma 11 [***WDT(M)*** $\wedge$ ***past-dominated(M)*** $\Rightarrow$***live(M)***] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) be a block machine derived from a consistent action diagram *AD*, such that the properties *WDT(M)* and *past-dominated(M)* are true. It follows that M is live.

Proof. To prove that M is live, we will show that it satisfies clauses P1 to P4 of Lemma 5. Clause P2 (i.e., each block of $\mathcal{B}$ has at least one trigger) is satisfied by definition of the trigger relation in a derived block machine (Definition 21). From the *consistent(AD)* and *WDT(M)* assumptions, and using the result of Lemma 10, it follows that clause P3 (i.e., the "$\prec$" relation on $\mathcal{B}$ is a partial order) is satisfied.

Consider an arbitrary (and possibly partial) execution of M, and let S be the sequence of blocks $[B_i]$, $i \geq 1$, enabled during that execution, where the blocks of S are in the order in which they were executed. Hence, this order implies an order in increasing block enabling time. Let $\mathcal{A}_i$ designate the set $trigs(B_i) \cup L_i$, where L_i is the set of local actions of block B_i. Let $\mathcal{A}^i$ be defined as:

$$\mathcal{A}^1 = \mathcal{A}_1$$
$$\mathcal{A}^i = \mathcal{A}^{i-1} \cup L_i \qquad \text{for } i > 1.$$

Let Z_i designate $Zone(\mathcal{A}_i, C_i)$, i.e., the zone in which the actions are the local actions and triggers of block B_i, and the constraints are the local constraints of B_i. Let Z^i be defined as:

$$Z^1 = Z_1$$
$$Z^i = Z^{i-1} \otimes Z_i \qquad \text{for } i > 1.$$

Let S^i designate the set $\{B_1, \ldots, B_i\}$ and V^i the vector of time assignments of actions of $\mathcal{A}^i$, as computed in the execution under consideration. In the following, we prove by induction on S, the three properties *InZone(i)*, *NoDeadlock(i)*, and *FTP(i)*, defined as follows:

- *InZone(i)* : $V^i \in Z^i$
- *NoDeadlock(i)* : the execution up to B_i has not entered the deadlock state
- *FTP(i)* : the execution up to B_i satisfies the forward time property

Induction base : Block B_1 (the first block of S) necessarily has the origin action as its only trigger, or else it could not be the first block to be enabled. Hence, Φ_1, the time computation function of B_1, chooses a vector of Z_1, such that the component of that vector corresponding to the origin takes on the value t_0.

From the *zone relativity* property (Lemma 6), such a vector exists, as long as Z_1 is not empty. It is indeed the case that Z_1 is not empty, since $CG(\mathcal{A}, C)$ is consistent. The existence of that vector implies *NoDeadlock(1)*, and the fact that the vector is in Z_1 means *inZone(1)*.

From *past-dominated(M)*, all tight paths between a pair of actions a_j, a_k are local to P_{jk} (where P_{jk} denotes *past*(a_j , a_k)). Hence, *dist*(a_j , a_k) can be computed over P_{jk} , i.e.,:

$$dist(a_j, a_k) = dist_{[P_{jk}]}(a_j, a_k) \tag{1}$$

Let a_1^k be an action of block B_1. From the *WDT(M)* assumption, we have *dist* (a_1^k, o) < 0 (see Definition 22). Using equation (1), and the fact that $P_{jk} = S^1$, we can rewrite the *WDT(M)* property on a_1^k, o as:

$$dist_{[S^1]}(a_1^k, o) < 0 \tag{2}$$

From *InZone(1)*, we know that the relationship between $t(a_1^k)$ and $t(o)$ (which are the occurrence times of a_1^k and o, respectively) is given by Z_1, which is the time zone defined by the actions and local constraints of S^1. As a result, relation (2) implies that the occurrence times of a_1^k and o in the execution are such that:

$$t(o) - t(a_1^k) < 0$$

Hence $t(a_1^k) > t(o)$. Since this is true for any action of B_1, it follows that B_1 satisfies the forward time property for the current execution, i.e., the property *FTP(1)* is true.

Induction step : Assume blocks B_1 to B_i have executed and B_{i+1} is the enabled block that is about to execute. Using Lemma 9, where Z_1, Z_2, $\mathcal{A}_1$ and $\mathcal{A}_{12}$ of Lemma 9 are Z^i, Z_{i+1}, $\mathcal{A}^i$ and *trigs*(B_{i+1}), respectively, we first show that:

$$Z^{i+1}\Downarrow_{\mathcal{A}^i} = Z^i \tag{3}$$

Indeed, If B_{i+1} has a single trigger (i.e., $|\mathcal{A}_{12}|$ of Lemma 9 is 1), then (3) follows directly from Lemma 9. Otherwise (i.e., B_{i+1} has more than one trigger), let a_j and a_k be a pair of triggers of B_{i+1}, $a_j \neq a_k$. Let P_{jk} denote *past*(a_j , a_k) and Q_{jk} denote $\mathcal{B} - P_{jk}$. From *past-dominated(M)*, we have:

$$dist_{[P_{jk}]}(a_j, a_k) < dist_{[Q_{jk}]}(a_j, a_k) \quad (4)$$

Since B_{i+1} is enabled and a_j, a_k are triggers of B_{i+1}, it must be that all blocks in P_{jk} have been enabled, and hence $P_{jk} \subseteq S^i$. Thus, S^i comprises at least all of the constraints of P_{jk}. As a result:

$$dist_{[S^i]}(a_j, a_k) \leq dist_{[P_{jk}]}(a_j, a_k) \quad (5)$$

Now, since the "$\prec$" relation on $\mathcal{B}$ is a partial order and since a_j and a_k are triggers of B_{i+1}, it follows that B_{i+1} cannot be in P, and hence it is in Q_{jk}. Thus, Q_{jk} comprises at least all of the constraints of B_{i+1}. As a result:

$$dist_{[Q_{jk}]}(a_j, a_k) \leq dist_{[B_{i+1}]}(a_j, a_k) \quad (6)$$

Combining (4) to (6), we get:

$$dist_{[S^i]}(a_j, a_k) < dist_{[B_{i+1}]}(a_j, a_k) \quad (7)$$

Since inequality (7) is true for an arbitrary pair a_j, a_k of *trigs*(B_{i+1}), it follows that it is true for all such pairs. Consequently, $Z^i \Downarrow_{trigs(B_{i+1})} \subseteq Z_{i+1} \Downarrow_{trigs(B_{i+1})}$, i.e., the premise of Lemma 9 holds, and hence equation (3) is true, indicating that all vectors of Z^i can be extended to B_{i+1}. Hence, from equation (3), and from the assumption that V^i is in Z^i (i.e., *inZone(i)*), it follows that V^i can be extended to B_{i+1}, and thus *NoDeadlock*(i+1) = *true*.

Designate by V^{i+1} the extension of V^i to B_{i+1}. Since V^i can be extended to B_{i+1}, and since V^i is in Z^i, it follows V^{i+1} is in $Z^i \otimes Z_{i+1}$, i.e., V^{i+1} is in Z^{i+1}, and hence *inZone*(i+1) is true.

Next, we show *FTP*(i+1). From *past-dominated(M)*, all tight paths between a pair of actions a_j, a_k are local to P_{jk} (where P_{jk} denotes *past*(a_j, a_k)). Hence, *dist*(a_j, a_k) can be computed over P_{jk}, i.e.,:

$$dist(a_j, a_k) = dist_{[P_{jk}]}(a_j, a_k) \quad (8)$$

Let a^k_{i+1} and $trig^j_{i+1}$ be an action and a trigger, respectively, of block B_{i+1} (where B_{i+1} is the block about to execute). From the *WDT(M)* assumption, we have $dist\,(a^k_{i+1}, trig^j_{i+1}) < 0$ (see Definition 22). Using equation (8), and the fact that $P_{jk} \subseteq S^{i+1}$, we can rewrite the *WDT(M)* property on a^k_{i+1}, $trig^j_{i+1}$ as:

$$dist_{[S^{i+1}]}(a^k_{i+1}, trig^j_{i+1}) < 0 \qquad (9)$$

From *InZone(i+1)*, we know that the relation between $t(a^k_{i+1})$ and $t(trig^j_{i+1})$ (which are the occurrence times of a^k_{i+1} and $trig^j_{i+1}$, respectively) is given by Z^{i+1}, which is the time zone defined by the actions and local constraints of S^{i+1}. As a result, inequality (9) implies that the occurrence times of a^k_{i+1} and $trig^j_{i+1}$ in the execution are such that:

$$t(trig^j_{i+1}) - t(a^k_{i+1}) < 0 \qquad (10)$$

Hence $t(a^k_{i+1}) > t(trig^j_{i+1})$. Since this is true for any action / trigger pair of B_{i+1}, it follows that B_{i+1} satisfies the forward time property in the execution. This and the inductive assumption *FTP(i)* implies *FTP(i+1)*.

Since *NoDeadlock(i)* and *FTP(i)* hold for every *i*, it follows that clauses P1 (M satisfies the forward time property) and P4 (no execution of M enters the deadlock state) of Lemma 5 hold. Hence, *live(M)* is true.

□

Consider, for example, the consistent action diagram of Figure 4. To satisfy Condition 1 of Definition 26, action o4 must be in a block all by itself. As for actions i2 and i3, they must be together in one block, or else the *WDT* condition would be violated (since i2 and i3 are concurrent to each other). Hence, the only block machine of interest that can be derived from AD_2 contains two action blocks, B_{in} and B_{out}, having the local action sets {i2, i3} and {o4}, respectively. Block B_{out} has two triggers, i2 and i3, which satisfy the *WDT* condition, since dist(o4, i2) < 0 and dist(o4, i2) < 0 (they are both equal to -11). B_{in} is triggered by the (implicit) origin action, and hence the *WDT* condition for B_{in} is satisfied by construction of the derived block machine (Definition 21). The machine satisfies the *past-dominated* property. Indeed, the shortest path from i2 to i3 is the edge (i2, i3) of weight 8 (associated with the constraint a1), and this path is local to *past*(i2, i3), since it is local to block B_{in}. The only other (i2, i3) path is of weight 9 and it consists of the edge (i2, o4) of weight 20, followed by the edge (o4, i3) of weight −11. The situation is symmetrical for (i3, i2) paths. As for the (i2, o4) and (o4, i2) paths, since *past*(o4) is the complete set of blocks, i.e., *past*(o4) = $\{B_{in}, B_{out}\}$, it follows that all paths, and in particular the (i2, o4) and (o4, i2) paths, are local to *past*(i2, o4). Similarly for the (i3, o4) and (o4, i3) paths, which are local to *past*(i3, o4). By Lemma 11, it follows that the machine is live.

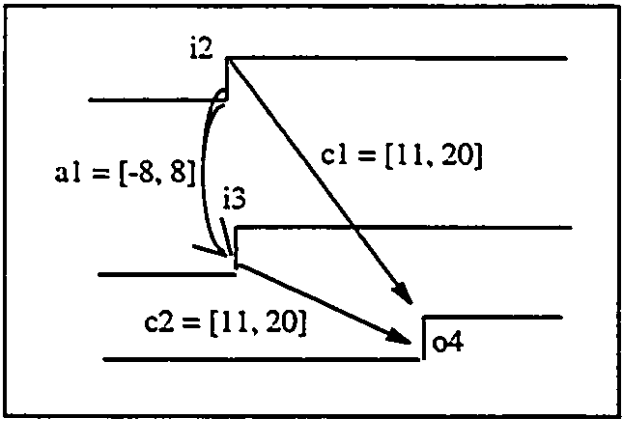


Figure 4: A causal action diagram.

Note that the *past-dominated(M)* condition is not, strictly speaking, a necessary condition for the liveness of *M*. The next lemma (Lemma 12) states that the conjunction of *WDT(M)* with a slightly weaker form of *past-dominated(M)*, designated as *weak-past-dominated(M)* (Definition 31), forms a *necessary* condition for the liveness of a block machine *M* derived from a consistent action diagram. We omit the proof of Lemma 12, because the *weak-past-dominated* criterion turns out to be of little practical interest (this is further discussed later in this section), and because in this paper we are more interested in the safety of our conditions than in their absolute minimality.

Definition 31 [*Weak-past-dominated(M)*] Let $M = (\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$ be a block machine derived from a consistent action diagram AD and satisfying the well-defined triggers property. M is said to satisfy the *weak past-dominated* property, denoted *weak-past-dominated(M)*, if:

$\forall\, a_i \in actions(\mathcal{A}), \forall\, a_j \in actions(\mathcal{A}) - \{a_i\}, \forall\, q_{ijk}$ tight path from a_i to a_j in $CG(\mathcal{A}, C)$,

q_{ijk} not local to $Past(a_i, a_j) \Rightarrow$

$\exists\, q_{ijl}$ tight path from a_i to a_j in $CG(\mathcal{A}, C)$, q_{ijl} local to $Past(a_i, a_j)$.

□

The *weak-past-dominated* condition requires that the tight paths that are not local to the past of an action pair be "backed up" by at least one tight path that is local to the past of the action pair. In other words, the tightest path local to the past must be tighter or, as tight as ($\leq$) the tightest path that is not local to the past. In contrast, the *past-dominated* condition (Definition 25) requires all tight paths between an action pair to be local to the past of this pair.

Lemma 12 [*live(M)* $\Rightarrow$ *WDT(M)* $\wedge$ *weak-past-dominated(M)*] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) be a live block machine derived from a consistent action diagram *AD*. It follows that the properties *WDT(M)* and *weak-past-dominated(M)* are true. □

Consider, for example, the action diagram AD_2 in Figure 3(b), which is the same as Figure 4 except that the assume constraint is of weight [-10, 10] (rather than [-8, 8] in Figure 4). For the same reasons as in the example of Figure 4, the only block machine derived from AD_2 that could be causal is the machine *M* with two action blocks, B_{in} and B_{out}, of local action sets {i2, i3} and {o4}, respectively. This machine, however, violates the *weak-past-dominated* property. Indeed, consider for example the tight (i2, i3) path p_{23} that consists of the edge e1 = (i2, o4) of weight 20, followed by the edge e2 = (o4, i3) of weight −11. The weight of p_{23} is 9 (= 20 − 11) and the constraints associated with the edges e1 and e2 are c1 and c2, respectively. These constraints are not local to *past*(i2, i3), and hence neither is p_{23}. Since there is no other (i2, i3) path that is as short as (or shorter than) p_{23}, it follows that the *weak-past-dominated* property is violated. To see that *M* is not live, consider for example the execution where t(i2) = 5 and t(i3) = 15 (which is allowed by the local constraints of B_{in}). Then, when B_{out} becomes enabled at T = 15, it will *not* be able to find a solution for t(o4) since that would require t(o4) to be greater than 26 (i.e., t(i3) + 11) and less than 25 (t(i2) + 20). Hence *M* enters the deadlock state, and thus it is non-live.

If we change the [−10, 10] assume constraint of Figure 3(b) to [−9, 9], the derived block machine described in the previous paragraph would satisfy the *weak-past-dominated* condition, but not the (stronger) *past-dominated* condition. In addition, the machine is now live: for any occurrence times of the actions i2 and i3 within the [−9, 9] assume constraint, block B_{out} is able to determine an occurrence time for o4 so as to satisfy all its local constraints (which are the two commit constraints of weight [11, 20]). Note, however, that

the (i2, i3) path that is local to *past*(i2, i3) has a weight equal to that of the (i2, i3) path that is *not* local to *past*(i2, i3). This has the effect that, in some executions of the machine, the function Φ of block B_{out} will return a single vector. Consider, for example, the execution where t(i2) = 5 and t(i3) = 14 (which is allowed by the local constraints of B_{in}). Then, when B_{out} becomes enabled at T = 15, it will find that there is only one solution for t(o4) that satisfies the local constraints of B_{out}; that solution is t(o4) = 25.

In terms of the practical framework of our application domain (specifications of asynchronous systems in a continuous time model), the kind of marginal situation outlined in the previous paragraph is of little practical interest, as it implies absolutely null design margins, thus making the specified system physically non-realizable, in practice. In view of these observations, we informally state that *past-dominated(M)* is an "almost necessary" condition for the liveness of a derived block machine M.

In terms of the theoretical framework, we have chosen the stronger *past-dominated* condition over its weaker counterpart as a liveness (and thus causality) criterion, because the stronger form has desirable compositional properties that allow us to express the compatibility of communicating action diagrams independently of the particular causal block machines that implement them. These compositional properties are put to advantage in the proof of Theorem 1 (the Compatibility Theorem). However, in order to do that, we will first need to rewrite the *past-dominated* condition into a provably equivalent form. This is the subject of the next section (Section 9).

9 Rewriting the *past-dominated* Condition

In this section, we show that the *past-dominated(M)* condition can be rewritten into a provably equivalent form, *loose-blocks(M)*, given in Definition 32. This rewriting enables us to prove the Compatibility Theorem (Theorem 1). An additional benefit of the *loose-blocks(M)* condition is that its computation is of time complexity $O(n^3)$, where n is the number of actions of M, whereas the worst case time complexity of *past-dominated(M)* could be exponential with n (it is based on path enumeration).

Definition 32 [***Loose-blocks***(M)] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) be a block machine derived from a consistent action diagram. M is said to satisfy the *loose blocks*

property, if:

$$\forall B_k \in \mathcal{B},\ \forall a_i \in trigs(B_k),\ \forall a_j \in trigs(B_k) - \{a_i\},$$
$$dist_{[CG(\mathcal{A}, C)]}(a_i, a_j) \neq \infty \implies dist_{[CG(\mathcal{A}, C)]}(a_i, a_j) < dist_{[B_k]}(a_i, a_j).$$

□

In Lemma 13 we shall establish the equivalence between the *loose-blocks* and the *past-dominated* properties of derived block machines. But first, a few definitions are in order.

Definition 33 [***Edge or path contained in a (set of) block(s)***] Let $M = (\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$ be a derived block machine and $Q \subseteq \mathcal{B}$. An edge e of $CG(\mathcal{A}, C)$ is said to be *contained* in Q, if there exists a block B_i in Q such that *source(e)* is local to B_i and there exists a block B_j in Q such that *sink(e)* is local to B_j (it does not matter whether $B_i \neq B_j$ or not). Furthermore, the edge e is said to be *contained* in B_i, if e is contained in $\{B_i\}$. A path r of $CG(\mathcal{A}, C)$ is said to be *contained* in Q (respectively B_i), if all the edges of r are contained in Q (respectively B_i).

□

Definition 34 [***Cross edge***] Let $M = (\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$ be a derived block machine. An edge $e_{ij} = (a_i, a_j)$ of $CG(\mathcal{A}, C)$ is a cross edge if a_i and a_j are not local to the same block, i.e.,: $\neg \exists B_k \in \mathcal{B}, a_i \in actions(B_k) \wedge a_j \in actions(B_k)$.

□

Definition 35 [***Direction of a cross edge***] Let $M = (\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$ be a derived block machine such that "$\prec$" on $\mathcal{B}$ (Definition 14) is a partial order. The ***direction*** of a cross edge $e_{ij} = (a_i, a_j)$ of $CG(\mathcal{A}, C)$ is one of ***right*** or ***left***, and is determined as follows, where $\mathcal{A}' = \mathcal{A} - \{o\}$.

1. If $a_i \in \mathcal{A}'$ and $a_j \in trigs(block(a_i))$, then e_{ij} is a ***left*** edge.
2. If $a_j \in \mathcal{A}'$ and $a_i \in trigs(block(a_j))$, then e_{ij} is a ***right*** edge.

□

In the above definition, the direction of each and every cross edge of the constraint graph of M is uniquely defined. Indeed, if both a_i and a_j are actions of $\mathcal{A}'$, then from the definition of the trigger relation of a derived block machine (Definition 21) and the assumption that "$\prec$" on $\mathcal{B}$ is a partial order, it follows

that the pre-condition of one and only one of Statements 1 and 2 of Definition 35 is true. Otherwise, one of a_i and a_j must be identical to the origin action o (they cannot both be identical to o because $a_i \neq a_j$). Since in a derived block machine there is no edge with o as its source, it follows that a_i cannot be equal to o. Hence, the only remaining possibility is $a_i \neq o$ and $a_j = o$, and therefore the pre-condition of Statement 2 of Definition 35 is false. From Definition 21, (a_i, o) is an edge of the constraint set of a derived block machine if, and only if o is a trigger of $block(a_i)$. Hence, the pre-condition of Statement 1 of Definition 35 is true. Therefore, in all cases, the direction of a cross edge is uniquely defined.

Definition 36 [*Transit*] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) be a derived block machine, $B_i \in \mathcal{B}$, and L_i the set of local actions of B_i. A *transit through* B_i, or *transit* for short, is a pair t = (*enter*, *exit*), where *enter* and *exit* are cross edges of $CG(\mathcal{A}, C)$, and:

- *enter* is such that: $source(enter) \in L_i$ and $sink(enter) \in L_i$,
- *exit* is such that: $source(exit) \in L_i$ and $sink(exit) \notin L_i$ and,
- there exists a path r of $CG(\mathcal{A}, C)$, such that $first(r) = enter$, $last(r) = exit$, and all other edges of r (if any) are contained in B_i. Any such path r is said to be a *path associated with the transit* t.

❑

Definition 37 [*Transit direction*] The direction of a transit t = (*enter*, *exit*) is a pair (*enterdir*, *exitdir*) where *enterdir* is the direction of the *enter* cross edge, and *exitdir* is the direction of the *exit* cross edge.

❑

Definition 38 [*Transit sequence*] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) be a derived block machine, $Q \subseteq \mathcal{B}$, and r a path of length ≥ 2, such that all edges of r, except the first and the last, are contained in Q. Then, the *transit sequence of r through Q*, is a uniquely defined sequence of transits $TS = [t_i,\ i = 1,\ldots, n]$, where $t_i = (enter_i, exit_i)$, $1 \leq i \leq n$, and such that:

- $enter_1 = first(r)$
- $enter_{i+1} = exit_i$, for $i = 1,\ldots, n-1$
- $exit(n) = last(r)$.

❑

Definition 39 [*Direction sequence*] Let $TS = [t_i,\ i = 1,\ldots, n]$ be the transit sequence of a path r through a set of blocks Q. The corresponding *direction sequence of r through Q* is the sequence of transit directions $DS = [dir_i,\ i = 1,\ldots, n]$, where $dir_i = (enterdir_i\ ,\ exitdir_i)$ is the transit direction of t_i, $i = 1,\ldots, n$. Obviously, $enterdir_{i+1} = exitdir_i,\ i = 1,\ldots, n-1$.

□

Lemma 13 [*loose-blocks(M)* $\Leftrightarrow$ *past-dominated(M)*] Let $M = (\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$ be a block machine derived from a consistent action diagram *AD*, such that "$<$" on $\mathcal{B}$ is a partial order. Then, *loose-blocks(M)* $\Leftrightarrow$ *past-dominated(M)*.

Proof ($\Rightarrow$). Let $CG = CG(\mathcal{A}, C)$. Assume that *loose-blocks(M)* is true and that *past-dominated(M)* is false, i.e., there exists a pair of actions a_i and a_j of $\mathcal{A}$ and a tight path s in *CG*, such that s is from a_i to a_j, an s is not local to past(a_i , a_j). In the following, we show that this leads to a contradiction.

Let P and Q designate $past(a_i\ ,\ a_j)$ and $\mathcal{B} - past(a_i\ ,\ a_j)$, respectively. Due to the assumption that s is not local to P, there must exist at least one edge in s that is not local to P. Such an edge must be local to Q. Consider the first such edge, e_1. Its source action $a_h = source(e_1)$ is in P or else e_1 would not be the first edge of s to be local to Q. The action $a_k = sink(e_1)$ cannot be in P or else e_1 would be local to P. It follows that a_k is in Q. In addition, a_k cannot terminate the path s, because s ends at a_j, with a_j in P, whereas a_k is in Q. Consequently, there must be at least one more edge in s following e_1. Since a_k is in Q and the termination of s is in P, there must exist at least one edge of s, after e_1, with its source in Q and its sink in P. Consider the first such edge, say e_p , and designate by r the subpath of s such that $r = e_1,\ldots, e_p$, $p \geq 2$. r is such that all its edges, except its first (i.e., e_1) and last (i.e., e_p) are contained in Q (or else e_p would not be the first edge of s with its source in Q and its sink in P). Thus, we can associate r with a transit sequence TS through Q, $TS = [t_i,\ i = 1,\ldots, n]$, $n \geq 1$, and $t_i = (enter_i, exit_i)$, $i = 1,\ldots,n$, such that $enter_1 = e_1$ and $exit_n = e_p$. Let $DS = [dir_i,\ i = 1,\ldots, n]$ be the direction sequence of r, where $dir_i = (enterdir_i\ ,\ exitdir_i)$ for $i = 1,\ldots, n$.

In the following, we show that e_1 is a *right* edge. Let $B_h = block(a_h)$, and $B_k = block(a_k)$. Since a_h is in P, a_k is in Q, and $P \cap Q = \emptyset$, it follows that a_h and a_k are necessarily in different action blocks, i.e., $B_k \neq B_h$. In addition, since e_1 is an edge from a_h to a_k, it follows that either a_h is a trigger of B_k, or a_k is a trigger

of B_h. However, the latter possibility, i.e., a_k trigger of B_h, would imply $B_k < B_h$, and consequently B_k would be in P (since B_h is in P, and P contains all blocks that are before B_h by the "$<$" relation on $\mathcal{B}$). This contradicts the assumption that B_k is in Q. Thus, the only possibility is that a_h is a trigger of B_k. This implies that $B_h < B_k$ and hence, $e_1 = (a_h , a_k)$ is a *right* edge. Since e_1 is the *enter* edge of the first transit in the TS transit sequence, it follows that, in TS, $enterdir_1 = right$.

Next we show that e_p is a *left* edge. Let $a_m = \text{sink}(e_p)$, $a_l = \text{source}(e_p)$, $B_m = \text{block}(a_m)$, and $B_l = \text{block}(a_l)$. Since a_m is in P and a_l is in Q, it follows that a_m and a_l are necessarily in different action blocks. In addition, since e_p is an edge from a_l to a_m, it follows that either a_m is a trigger of B_l, or a_l is a trigger of B_m. However, the latter possibility, i.e., a_l trigger of B_m, would imply $B_l < B_m$, and consequently B_l would be in P (since B_m is in P, and P contains all blocks that are before B_h by the "$<$" relation on $\mathcal{B}$). This contradicts the assumption that B_l is in Q. Thus, the only possibility is thus that a_m is a trigger of B_l. This implies that $B_m < B_l$ and hence, $e_p = (a_l, a_m)$ is a *left* edge. Since e_p is the *exit* edge of the *nth* (and last) transit in the TS transit sequence, it follows that, in TS, $exitdir_n = left$.

So, we have: $enterdir_1 = right$, $enterdir_{i+1} = exitdir_i$, for $i = 1,\ldots, n-1$, and $exitdir_n = left$. In the following, we show that there must exist k, $1 \le k \le n$, such that $dir_k = (right , left)$, i.e., $enterdir_k = right$ and $exitdir_k = left$. We first show by induction on i that $prop_i$ is true for all i, $i = 1, \ldots n$, where $prop_i = [\exists\, k \le i , dir_k = (right , left)] \vee [\forall\, k \le i , dir_k = (right , right)]$.

Induction base: Since $enterdir_1 = right$, it follows that either $dir_1 = (right, left)$, or $dir_1 = (right, right)$. Thus $prop_1$ is true.

Induction step: If the first clause of the disjunction in $prop_i$ is true, it trivially follows that the first clause of the disjunction of $prop_{i+1}$ is also true, and thus $prop_{i+1}$ is true. If the second clause of the disjunction of $prop_i$ is true, it follows that $enterdir_{i+1} = right$, because $enterdir_{i+1} = exitdir_i$, for $i = 1,\ldots, n-1$. In that case, there are two possibilities for dir_{i+1}: either it is $(right , left)$, or it is $(right , right)$. The first possibility makes the first clause of $prop_{i+1}$ true, and the second possibility makes the second clause of $prop_{i+1}$ true. Thus, in all cases, $prop_i \Rightarrow prop_{i+1}$, and hence, by the induction principle, $prop_i$ is true for all i, $1 \le i \le n$.

Therefore, for $i = n$, $prop_n = [\exists k \le n, dir_k = (right, left)] \vee [\forall k \le n, dir_k = (right, right)]$ is true. Since $exitdir_n = left$, the second clause of the disjunction of $prop_n$ is false, and thus the only possibility is the first clause, i.e., there exits $k \le n$, $dir_k = (right, left)$. Consider one such k, and let t_k be the k^{th} transit in TS, i.e., corresponding to dir_k. Let $\bar{B}$ be the block through which t_k transits, a_y and a_z be the source of the $enter_k$ edge and the sink of the $exit_k$ edge, respectively. Since $enter_k$ is a *right* edge, it follows that a_y is a trigger of $\bar{B}$. Since $exit_k$ is a *left* edge, it follows that a_z is a trigger of $\bar{B}$. In addition, since (1) by assumption, r is a tight path, (2) all subpaths of a tight path are tight paths, and (3) t_k is a transit in the transit sequence associated with r, it follows that there must be at least on tight path associated with the transit t_k through block $\bar{B}$. Hence, $dist_{[CG]}(a_y, a_z) = dist_{[\bar{B}]}(a_y, a_z)$. This implies that $\bar{B}$ violates the *loose-block* property, which contradicts the initial assumption.

Proof ($\Leftarrow$). Assume that *past-dominated(M)* is true and that *loose-blocks(M)* is false, i.e., there exists a block $\bar{B}$ and a pair of triggers a_y, a_z of $\bar{B}$, such that $dist_{[\bar{B}]}(a_y, a_z)$ is finite and $dist_{[CG]}(a_y, a_z) = dist_{[\bar{B}]}(a_y, a_z)$. This means that there is a tight path q from a_y to a_z, and q is local to $\bar{B}$. In addition, since a_y and a_z are triggers of $\bar{B}$, and since the "<" relation on $\mathcal{B}$ is a partial order, it follows that $\bar{B} \notin past(a_y)$ and $\bar{B} \notin past(a_z)$. As a result, $\bar{B} \in past(a_y, a_z)$. Now, since q is local to $\bar{B}$, and since $\bar{B} \in past(a_y, a_z)$, it follows that the tight path q is not local to $past(a_y, a_z)$. This contradicts the *past-dominated(M)* assumption. ❑

For example, Figure 5 shows the action diagram and an action partition of the READ cycle of the Motorola MC68360 processor. Blocks are delimited using dashed lines; e.g., the trigger of block EB_{11} is the (implicit) *origin* action. AS↓ is the only trigger of block EB_{12}, actions CK↑2 in EB_{11} and ACK↓ are triggers of EB_{15}, etc. All the conditions of Definition 26, wherein *loose-blocks*(M) is substituted for the *past-dominated*(M) condition, are satisfied. Hence, by Definition 27, the action diagram is causal. Similarly, Figure 6 depicts a read cycle with a causal action partition of a slave device that could be connected to the processor of Figure 5.

10 Trace Set Conservation

In this section, we prove that the trace set of a causal derived block machine is equal to the trace set of the action diagram from which it was derived.

Lemma 14 [$M \in dBMs(AD) \wedge WDT(M) \Rightarrow TraceSet(AD) \subseteq TraceSet(M)$] Let M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) be a block machine derived from a consistent action diagram AD. Then, if M satisfies the well-defined trigger property (Definition 22), it follows that $TraceSet(AD) \subseteq TraceSet(M)$.

Proof. Consider Ω a trace of AD, and let V be the vector of action occurrence times corresponding to Ω. Let $Z_{AD} = Zone(\mathcal{A}, C)$ be the global zone of AD. The fact that Ω is a trace of AD is equivalent to saying that $V \in Z_{AD}$. Let $\mathcal{A} = \{a_0, \ldots, a_n\}$, such that $a_0 = o$, and let τ_j designate the occurrence time in V of an action a_j of $\mathcal{A}$, $0 \le j \le n$. Let $S = [B_i]$, $i = 1,\ldots, n$, be a sequence of blocks sorted in increasing Θ_i (with arbitrary order amongst blocks that have the same Θ_i), where Θ_i is defined in the following (with $trigs(B_i)$ being the set of triggers of a block B_i):

$$\Theta_i = \operatorname*{Max}_{a_k \in trigs(B_i)}(\tau_k) \tag{11}$$

In order to prove that Ω is a trace of M, it suffices to "construct" an execution E of M, such that E satisfies the property $\forall i\ Prop_i$, where:

- $Prop_i$: $AgendaSimulate_i(S) \wedge enabled_i \wedge NoDeadlock_i \wedge BlockSimulate_i(V) \wedge noIndefiniteWait_i$
- $AgendaSimulate_i(S)$: when all blocks preceding B_i in the sequence S have executed in E, the following block to execute in E is B_i.
- $enabled_i$: block B_i is enabled in the execution E at time Θ_i.
- $NoDeadlock_i$: when block B_i has executed in E, Φ_i (the time computation function of B_i) returns a non-empty set.
- $BlockSimulate_i(V)$: when block B_i has executed in E, the *choose* function invoked by the execution (Definition 16) returns $V\Downarrow_{L_i}$.
- $noIndefiniteWait_i$: once block B_i has executed in E, the local actions of B_i cannot cause the *WAIT* operator in the execution E to remain in a *wait* state for an infinite amount of time.

In the rest of the proof of this lemma, we use the notation L_i, $\mathcal{A}_i$, Z_i, $\mathcal{A}^i$, and S^i that was defined in the proof of Lemma 11. Let $t(trigs(B_i))$ designate the

occurrence time vector, in the execution E, of the triggers of block B_i. In the following, we show that, for an arbitrary block B_i :

$$[enabled_i \wedge t(trigs(B_i)) = V\Downarrow_{trigs(B_i)}] \Rightarrow NoDeadlock_i \wedge BlockSimulate_i(V) \wedge noIndefiniteWait_i \quad (12)$$

Since $\mathcal{A}_i \subseteq \mathcal{A}$ and $C_i \subseteq C$, it follows that $Z_{AD}\Downarrow_{\mathcal{A}_i} \subseteq Z_i$. This, together with the fact that $V \in Z_{AD}$, implies that $V\Downarrow_{\mathcal{A}_i} \in Z_i$. The latter statement is itself equivalent to $V\Downarrow_{L_i} \in \Phi_i(V\Downarrow_{trigs(B_i)})$, where Φ_i is the time computation function of B_i.

The fact that $V\Downarrow_{L_i} \in \Phi_i(V\Downarrow_{trigs(B_i)})$ implies that *if* B_i is enabled and *if* its trigger time vector, $t(trigs(B_i))$, is $V\Downarrow_{trigs(B_i)}$, then the first consequence is that $\Phi_i(V\Downarrow_{trigs(B_i)})$ is not empty, and hence $NoDeadlock_i$, and the second consequence is that $V\Downarrow_{L_i}$ is part of the choices that the block machine can make for the local action times of B_i. Thus, we can make $V\Downarrow_{L_i}$ be the chosen occurrence time vector for the actions of B_i in E, and hence we have $BlockSimulate_i(V)$. In addition, from *WDT(M)*, we have: $\forall\, a_j \in trigs(B_i)$, $\forall\, a_k \in actions(B_i)$, $\tau_k > \tau_j$. Hence, $\tau_k > \Theta_i$. Since M satisfies the *WDT* property, it follows that: $\forall\, a_k \in actions(B_i)$, $\tau_k > \Theta_i$. Combining this with the fact that block B_i is enabled at $T = \Theta_i$ (consequence of the $enabled_i$ assumption) and with the fact that a block executes at the time when it is enabled (see Definition 16), it follows that, when Φ_i is evaluated, all components of all vectors computed by Φ_i are strictly greater than the current value of T (the current time). Hence, no local action of B_i can be the cause of an indefinite suspension of the WAIT operator in the execution of the block machine, i.e., $NoIndefiniteWait_i$. Thus, (12) holds.

In the following, we show by induction on i that $\forall i$, $Prop_i$.

Induction base : From the definition of $Z_{AD} = Zone(\mathcal{A}, C)$, there is a unique action $a_i \in \mathcal{A}$, such that for each and every action $a_j \in \mathcal{A}$, $a_j \neq a_i$, the property $dist(a_j , a_i) < 0$ holds. In addition, this unique action a_i is the origin action o. Hence, if τ_0 is the time in V of the origin o, it follows that:

$$\forall j\ a_j \in \mathcal{A} \Rightarrow \tau_0 < \tau_j \quad (13)$$

Let $\mathcal{T}^*$ be the restriction of $\mathcal{T}$ to $\mathcal{A} \times \mathcal{B}$ (recall: $\mathcal{T}$ is defined on $\mathcal{A} \times \mathcal{B}$). If all blocks of $\mathcal{B}$ were covered by $\mathcal{T}^*$ (i.e., if all blocks of $\mathcal{B}$ had at least one trigger in $\mathcal{A}$), then the "$<$" relation on $\mathcal{B}$ would not be a partial order, thus contradict-

ing Lemma 10. Hence, the set $\mathcal{B}_0$, defined as the set of blocks that have no triggers in $\mathcal{A}$, is not empty. From Definition 21, all blocks of $\mathcal{B}_0$ are assigned the origin o as their only trigger. Hence, for any block B_k of $\mathcal{B}_0$, we have $\Theta_k = V_{\Downarrow\{o\}} = \tau_0$, and the blocks of $\mathcal{B}_0$ are the only ones that have an Θ value equal to τ_0. Hence, using (13), it follows that the blocks of $\mathcal{B}_0$ share a single unique minimum Θ. Since B_1 (the first block in S) is a block with minimum Θ, and since $\mathcal{B}_0$ is not empty, it follows that B_1 is a member of $\mathcal{B}_0$ and hence B_1 has the origin as only trigger.

From the execution model (Definition 16), blocks which have the origin as the only trigger are enabled at time t_0, where t_0 is the parameter of the procedure M_{exec}. Hence, if we choose τ_0 for the parameter t_0, we get that B_1 is enabled at τ_0, and thus $enabled_1$ is true. Since, in addition to $enabled_1$, we have $t(trigs(B_1)) = V_{\Downarrow trigs(B_1)} = \tau_0$, using (12) we conclude that $NoDeadlock_1 \wedge BlockSimulate_1(V) \wedge noIndefiniteWait_1$.

Since B_1 is enabled at τ_0, there is certainly an execution in which B_1 executes before other blocks. Let E be that execution. (Note: in the case where there are other blocks that are also enabled at τ_0, then other executions might choose some of these other blocks to execute first). Since there are no blocks preceding B_1 in the sequence S, it follows that the property $AgendaSimulate_1$ holds. As a result, $Prop_1$ is true.

Induction step : The induction hypothesis is that, for some i such that $i < n$ (where n is the length of S), $Prop_j$ holds, for all j such that $j \leq i$. We need to show that this implies that $Prop_{i+1}$ holds too.

In the following, we first show that B_{i+1} is triggered only by actions of $\mathcal{A}^i$. Consider a block B_m, where $m \geq i+1$. From the *WDT(M)* property, we have (where L_m is the set of local actions of B_m, and τ_k is the component of V corresponding to action a_k) : $\forall\, a_k \in L_m\,,\, \tau_k > \Theta_m$. In addition, since the block sequence S is sorted in increasing Θ, it follows that: $\Theta_m \geq \Theta_{i+1}$. Hence:

$$\forall\, a_k \in L_m\,,\, \tau_k > \Theta_{i+1}.$$

From the expression (11) of Θ_{i+1}, this implies that:

$$\forall a_l \in trigs(B_{i+1}),\, \forall a_k \in L_m\,,\, \tau_k > \tau_l\,.$$

Thus, the two sets $\{a_l \mid (a_l , B_{i+1}) \in \mathcal{T}\}$ and L_m are disjoint, and hence, no action of block B_m, for any m, $m \geq i+1$, can be a trigger of block B_{i+1}. As a result, B_{i+1} is triggered only by actions of $\mathcal{A}^i$.

The execution context for the induction step is that blocks B_1 to B_i have been evaluated. The execution has just finished evaluating block B_i. Let $\mathcal{H}_i$ be the set of blocks that have not executed yet, i.e., $\mathcal{H}_i = S - S^i$. Let $\mathcal{K}_i$ be the subset of blocks of $\mathcal{H}_i$ that have all their triggers in $\mathcal{A}^i$ and let $\mathcal{L}_i$ be the set $\mathcal{H}_i - \mathcal{K}_i$. From the inductive assumptions $enabled_j$ and $NoDeadlock_j$, we know that all blocks B_j such that $j \leq i$, have executed and have a non-empty Φ_j solution. Thus, at this point of the execution, the enabling times of all blocks of $\mathcal{K}_i$ are known. Due to the inductive assumption $BlockSimulate_j(V)$, for all j such that $j \leq i$, we also know that if a block of $\mathcal{K}_i$ is actually enabled, then its enabling time is equal to its Θ value. From the inductive assumption $noIndefiniteWait_j$, for all j such that $j \leq i$, we know that no action of $\mathcal{A}^i$ can cause the execution algorithm (Definition 16) to remain forever in a *WAIT* state. Thus, up and until a subsequent block is enabled, nothing can prevent $\mathcal{A}^i$ actions to occur at their respective computed occurrence times.

Consider a block B' of $\mathcal{L}_i$. By definition of being in $\mathcal{L}_i$, B' has at least one trigger which is in $\mathcal{H}_i$, i.e., this trigger is local to a block that has not been evaluated yet. Hence, in order for B' to be evaluated, at least one other block has to be evaluated first. As for blocks in $\mathcal{K}_i$, since all their triggers are in $\mathcal{A}^i$, they require no blocks to be evaluated as a pre-condition for their own evaluation. In addition, $\mathcal{K}_i$ is not empty, since it contains at least B_{i+1} (which was shown above to be triggered only by actions of $\mathcal{A}^i$). Hence, there exists at least one block in $\mathcal{K}_i$ that will be evaluated before all blocks of $\mathcal{L}_i$. In addition, the first such $\mathcal{K}_i$ block to be evaluated is necessarily a block with smallest enabling time amongst the $\mathcal{K}_i$ blocks. B_{i+1} is such a block, since the enabling times of $\mathcal{K}_i$ blocks are equal to their Θ value, and the block sequence S is sorted in increasing Θ values. Thus, up and until B_{i+1} is enabled, $\mathcal{A}^i$ actions occur at their computed occurrence time. Hence, B_{i+1} is indeed enabled, i.e., $enabled_{i+1}$. And since B_{i+1} is the next enabled block (or among a set of blocks to be next enabled simultaneously), there exists an extension of the execution E in which the next block to be evaluated is B_{i+1}. Thus, $AgendaSimulate_{i+1}(S)$.

Finally, from the inductive assumption $BlockSimulate_j(V)$, for all j such that $j \leq i$, we know that the computed occurrence time vector of the actions of each

such block B_j is $V\Downarrow_{L_j}$. In addition, since every trigger of B_{i+1} is local to some B_j with $j \le i$, it follows that $t(trigs(B_{i+1})) = V\Downarrow_{trigs(B_{i+1})}$. Using (12), this implies that $NoDeadlock_{i+1} \wedge BlockSimulate_{i+1}(V) \wedge noIndefiniteWait_{i+1}$. Hence, $Prop_{i+1}$ holds.

□

Lemma 15 [$M \in dBMs(AD) \wedge WDT(M) \wedge past\text{-}dominated(M)$
$\Rightarrow TraceSet(M) = TraceSet(AD)$]

Let AD = (S, $\mathcal{A}$, o, C) be a consistent action diagram and M = ($\mathcal{A}$, o, $\mathcal{B}$, $\mathcal{T}$, C) a block machine derived from AD, such that M satisfies the well-defined triggers (Definition 22) and past dominated (Definition 25) properties. Then, $TraceSet(AD) = TraceSet(M)$.

Proof. From Lemma 11, we know that M is live and that, for an arbitrary execution E with execution vector V^n, the property $V^n \in Z^n$ holds (using the terminology of Lemma 11). Since $Z^n = Z_{AD}$ (where Z_{AD} is the global zone of AD, i.e., $Z_{AD} = Zone(\mathcal{A}, C)$), it follows that $V^n \in Z_{AD}$, and hence V^n is a trace of AD. Thus, $TraceSet(M) \subseteq TraceSet(AD)$. In addition, from Lemma 14, we have $TraceSet(AD) \subseteq TraceSet(M)$. Hence, $TraceSet(AD) = TraceSet(M)$.

□

The implication of Lemma 15 is that an action diagram is either non-causal in our sense of the word (when no causal block machine can be derived from it), or else all its possible interpretations "that make sense", i.e., all causal block machines derived from it, are trace equivalent. The existence of multiple equivalent block machines can be important in the synthesis of interface controllers, for example when exploring implementation alternatives with different block granularity and degrees of control distribution, and when selecting solutions that satisfy various design requirements. Such considerations are, however, beyond the scope of this paper.

11 Compatibility of Communicating Action Diagrams

In this section, we develop a procedure for verifying whether a set of communicating causal action diagrams are compatible, i.e., whether *any* combination of their derived causal block machines are compatible. First, we formalize the concepts of *connection* (Definition 40 and Definition 41), *composition*

(Definition 43 and Definition 44) and *compatibility* (Definition 45 and Definition 46). Then, in Theorem 1, we prove that we do *not* need to enumerate the combinations of derived block machines in order to answer the action diagram compatibility question. The theorem provides an exact and efficient procedure for the verification of the compatibility of communicating action diagrams.

The block machines $M_1,\ldots, M_n$ and action diagrams $AD_1,\ldots, AD_n$ under consideration are defined on distinct action sets, $\mathcal{A}_1, \ldots, \mathcal{A}_n$, i.e., $i \neq j$ implies that $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$, for $i = 1, \ldots, n$, $j = 1, \ldots, n$. We also assume that the port sets $S_1,\ldots, S_n$, of the action diagrams are distinct, i.e., $i \neq j$ implies that $S_i \cap S_j = \emptyset$ for $i = 1, \ldots, n$, $j = 1, \ldots, n$.

Definition 40 [***Port connection group***] Let $Q = \{AD_1, \ldots, AD_n\}$ be a set of action diagrams with $AD_i = (S_i, \mathcal{A}_i, o_i, C_i)$ and let $\Sigma_S = \bigcup_{i=1}^{n} S_i$. Then:

- A *port connection* over Q is a pair $PCon = (P, PortSet)$, where P is a port such that $P \notin \Sigma_S$ and $Portset \subseteq \Sigma_S$. P is said to be the *communication port* of *PCon*, and *PortSet* is said to be the *port set* of *PCon*. In addition, for any port P' of *PortSet*, P is said to be the communication port corresponding to P'.
- A *port connection group* over Q is a pair $PConG = (GPortSet, PConSet)$, where *PConSet* is a set of port connections over Q, $PConSet = \{PCon_j \mid PCon_j = (P_j, PortSet_j), j = 1,\ldots, m\}$ and $GPortSet = \{P_j \mid j = 1,\ldots, m\}$.
- A port connection group *PConG* over Q is ***sound*** if:
  - Each port of each AD_i of Q is an element of the *PortSet* of one and only one port connection $PCon_j$ of *PConG*.
  - The number of output ports in the *PortSet* of each port connection of *PConG* is exactly one.
  - All ports of a *PortSet* of any given port connection must have the same number of actions.
- Given a sound port connection group $PConG = (GPortSet, PConSet)$, let $PCon_j$ be a port connection of *PConSet*, with $PCon_j = (P_j, PortSet_j)$, and let P_j' be the unique output port of $PortSet_j$, and AD_k be the action diagram for which P_j' is an output port, i.e., $P_j' \in S_k$. Then, the *direction* of P_j is k.

□

Note that in the above definition, the "direction" of a (communication) port is an integer from 1 to n that identifies the action diagram that controls the port. This is a slight generalization of Definition 2 in which the direction identified which of the environment (*in*) or device (*out*) controls a port. In the next definition, we carry on this generalization to the direction of *communication actions*.

Definition 41 [*Action connection group*] Consider a set $Q = \{X_1, \ldots, X_n\}$ of either n action diagrams, or n block machines. Let $\mathcal{A}_i$ and o_i be the action set and the origin action, respectively, of X_i and let $\Sigma_{\mathcal{A}} = \bigcup_{i=1}^{n} \mathcal{A}_i$. Then:

- An *action connection* over Q is a pair $ActCon = (a, ActSet)$, where a is an action such that $a \notin \Sigma_{\mathcal{A}}$ and $ActSet \subseteq \Sigma_{\mathcal{A}}$. a is said to be the *communication action* of *ActCon*, and *ActSet* is said to be the *action set* of *ActCon*. In addition, for any action a' of *ActSet*, a is said to be the communication action corresponding to a'.
- An *action connection group* over Q is a pair $ActConG = (GActSet, ActConSet)$, where *ActConSet* is a set of action connections over Q, $ActConSet = \{ActCon_j \mid ActCon_j = (a_j, ActSet_j),\ j = 1,\ldots, m\}$ and $GActSet = \{a_j \mid j = 1,\ldots, m\}$.
- An action connection group *ActConG* over Q, $ActConG = (GActSet, ActConSet)$, is ***sound*** if:
  - Each action of each $\mathcal{A}_i$, $i = 1,\ldots, n$, is an element of the *ActSet* of one and only one action connection $ActCon_j$ of *ConG*.
  - There exists an action o of *GActSet*, such that o is of *null* direction and the connection $(o, \{o_1, \ldots, o_n\})$ is a member of *ActConSet*. This connection is designated as the ***origin connection***.
  - For any action connection $ActCon_j$, other than the origin connection, the number of output actions in $ActSet_j$ is one.
- Given a sound action connection group $ActCon = (GActSet, ActConSet)$, let $ActCon_j$ be an action connection of *ActConSet*, other than the origin connection, with $ActCon_j = (a_j, ActSet_j)$, and let $a_j^{\cdot}$ be the unique output action of $ActSet_j$, and X_k be the element of Q for which $a_j^{\cdot}$ is an output action, i.e., $a_j^{\cdot} \in \mathcal{A}_k$. Then, the direction of a_j is k.

□

Definition 42 [*Derived action connection group*] Consider $Q = \{AD_1, \ldots, AD_n\}$ a set of action diagrams, where $AD_i = (S_i, \mathcal{A}_i, o_i, C_i)$. Let *PConG* be a *sound* port connection group over Q, with $PConG = (GPortSet, PConSet)$ and with $PConSet = \{PCon_j \mid PCon_j = (P_j, PortSet_j), j = 1, \ldots, m\}$. Let $\Sigma_{\mathcal{A}}$ designate $\bigcup_{i=1}^{n} \mathcal{A}_i$. Then, the derived action connection group, $ActConG = derived\text{-}Con(PConG, \{o_1, \ldots, o_n\})$ is such that $ActConG = (GActSet, ActConSet)$, and:

- *ActConSet* is composed of the following action connections:
  - $(o, \{o_1, \ldots, o_n\})$, where $o \notin \Sigma_{\mathcal{A}}$, and o is of the *null* direction.
  - each port connection $PCon_j$ of *PConG*, where $PCon_j = (P_j, PortSet_j)$, "derives" m_j action connections, where m_j is the number of actions of a port of $PortSet_j$. The k^{th} of these m_j action connections, for any k, $1 \le k \le m_j$, is $ActCon_{jk} = (a_{jk}, ActSet_{jk})$, where $a_{jk} \notin \Sigma_{\mathcal{A}}$, and $ActSet_{jk}$ is the set of k^{th} actions of the ports of $PortSet_j$. The sequence $[a_{jk}]$, $k = 1, \ldots, m_j$, is said to be the ***communication action sequence*** of P_j. The direction of each action of the sequence $[a_{jk}]$ is that of P_j.
- $GActSet = \{o\} \cup \left(\bigcup_{j=1}^{m} \bigcup_{k=1}^{m_j} a_{jk} \right)$.

□

Referring to Definition 42, clearly, if *PConG* is sound, then so is *ActConG*. In the following, we generalize the notion of constraint intent to integers in a similar way to the generalization of port and action directions. These generalizations allow us to define the composition of action diagrams as yielding a structure which itself is an action diagram, thus allowing the re-use of previously proven results.

Definition 43 [***Action Diagram Composition***] Consider $Q = \{AD_1, \ldots, AD_n\}$ a set of action diagrams, where $AD_i = (S_i, \mathcal{A}_i, o_i, C_i)$. Let *PConG* be a *sound* port connection group over Q, with $PConG = (S, PConSet)$, and $ActConG = (\mathcal{A}, ActConSet) = derivedCon(PConG, \{o_1, \ldots, o_n\})$. Then, the composed action diagram $AD = composed\text{-}AD(PConG, Q)$ is $AD = (S, \mathcal{A}, o, C)$, where:

- o is the communication action of the action connection $(o, \{o_1, \ldots, o_n\})$.
- The action sequence of any port of S is its *communication action sequence* (Definition 42).

- C is obtained by first taking the union of the set of *commit* constraints of $AD_1, \ldots, AD_n$, and then substituting a constraint $c' = (a_i', a_j', \pi)$ for each constraint $c = (a_i, a_j, \pi)$ of the resulting set, where a_i' (respectively a_j') is the communication action corresponding to a_i (respectively a_j) in *ActConG*. If AD_k is the action diagram for which c is a commit constraint, then the intent of c' is k. In substituting c' for c, we say that constraint c is ***transposed*** to *ActSet*.

□

Lemma 16 [***Inclusion of composition zone***] Consider a set Q of consistent action diagrams, $Q = \{AD_1, \ldots, AD_n\}$, where $AD_i = (S_i, \mathcal{A}_i, o_i, C_i)$, $i = 1, \ldots, n$. Let *PCon* be a sound port connection group over Q, and let $\mathcal{A}$ be the set of communication actions of the action connection group derived from *PCon*. Let C_{iC} (respectively C_{iA}) be the set of commit constraints (respectively assume constraints) of AD_i transposed to $\mathcal{A}$. Let $C_C = C_{1C} \cup \ldots \cup C_{nC}$ and $C_A = C_{1A} \cup \ldots \cup C_{nA}$. Then, if $Zone(\mathcal{A}, C_C)$ is non-empty and $Zone(\mathcal{A}, C_C) \subseteq Zone(\mathcal{A}, C_A)$, it follows that $Zone(\mathcal{A}, C_C) \subseteq Zone(\mathcal{A}, C_{iC} \cup C_{iA})$, for $i = 1, \ldots, n$.

Proof. Since C_{iA} is a subset of C_A, it follows that $Zone(\mathcal{A}, C_A) \subseteq Zone(\mathcal{A}, C_{iA})$. In addition, by the lemma assumption, $Zone(\mathcal{A}, C_C) \subseteq Zone(\mathcal{A}, C_A)$. Hence:

$$Zone(\mathcal{A}, C_C) \subseteq Zone(\mathcal{A}, C_{iA}) \tag{14}$$

(14) implies that:

$$Zone(\mathcal{A}, C_C) = Zone(\mathcal{A}, C_C \cup C_{iA}) \tag{15}$$

Since $Zone(\mathcal{A}, C_C \cup C_{iA})$ contains all of the constraints of $Zone(\mathcal{A}, C_{iC} \cup C_{iA})$, it follows that:

$$Zone(\mathcal{A}, C_C \cup C_{iA}) \subseteq Zone(\mathcal{A}, C_{iC} \cup C_{iA}) \tag{16}$$

From (15) and (16), we obtain that $Zone(\mathcal{A}, C_C) \subseteq Zone(\mathcal{A}, C_{iC} \cup C_{iA})$.

□

Definition 44 [*Block Machine Composition*] Let Q be a set of block machines, $Q = \{M_1, \ldots, M_n\}$, where $M_i = (\mathcal{A}_i, o_i, \mathcal{B}_i, \mathcal{T}_i, C_i)$ is derived from a consistent action diagram AD_i, $i = 1, \ldots, n$, and let $ActConG = (\mathcal{A}, ActConSet)$ be a sound action connection group over Q. We define the composed block machine $M = composed\text{-}BM(ActConG, Q)$, as follows. $M = (\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$, where:

- o is the unique action of *null* direction in *ActSet*.
- $\mathcal{B}$ is obtained by first taking the union of the *output* blocks of M_i, for all i, $i = 1, \ldots, n$, and then replacing each block B_{ik} in the resulting set by a block B'_{ik} such that the set of local actions of B'_{ik} is the set of communication actions corresponding to the local actions of B_{ik}.
- $\mathcal{T}$ is obtained by first taking the union of the trigger relations $\mathcal{T}_i$ restricted to the output blocks of M_i, for all i, $i = 1, \ldots, n$, and then replacing, in the resulting set, each pair (a_{ij}, B_{ik}) that originates from some $\mathcal{T}_i$, by (a'_{ij}, B'_{ik}), where a'_{ij} is the communication action corresponding to a_{ij}, and B'_{ik} is the block of $\mathcal{B}$ that was substituted for B_{ik}.
- C is the union of the commit constraints of $M_1, \ldots, M_n$, transposed to *ActSet*.

❑

Definition 45 [*Compatible block machines*] Consider $Q = \{M_1, \ldots, M_n\}$ a set of causal action diagrams and $ActConG = (ActSet, ActConSet)$ a sound action connection group over Q. Let C_{iA} be the set of assume constraints of M_i transposed to *ActSet*, and let $C_A = C_{1A} \cup \ldots \cup C_{nA}$. Then, the action diagrams $M_1, \ldots, M_n$ are said to be ***compatible*** with respect to *ActConG*, written $compatible(ActConG, \{M_1, \ldots, M_n\})$, if:

$composed\text{-}BM(ActConG, \{M_1, \ldots, M_n\})$ is causal
$\wedge$ $TraceSet(composed\text{-}BM(ActConG, \{M_1, \ldots, M_n\}))$ *satisfies* C_A.

❑

In other words, the first condition for block machine compatibility is that the composed machine be causal. This essentially means (from Definition 26 and Lemma 11) that the collective behavior of the interconnected machines must be live. The second condition is that all the executions of the composed machine must satisfy all the assume constraints of all the interconnected machines. In the next definition, we state the criterion for action diagram compatibility in

such a way that its satisfaction guarantees block machine compatibility, irrelevant of which causal machine combination is chosen.

Definition 46 [***Compatible action diagrams***] Consider $Q = \{AD_1,..., AD_n\}$ a set of causal action diagrams, *PConG* a *sound* port connection group over Q, and $ActConG = (ActSet, ActConSet)$ the sound action connection group derived from $(PConG, \{o_1,..., o_n\})$. Let C_{iA} be the set of assume constraints of AD_i transposed to *ActSet*, and $C_A = C_{1A} \cup ... \cup C_{nA}$. Then, $AD_1,..., AD_n$ are said to be ***compatible*** with respect to *PConG*, written $compatible(PConG, \{AD_1,..., AD_n\})$, if: $\forall AD_i \in Q, \forall M_{ij_i} \in CdBMs(AD_i)$,

$$compatible(ActConG, \{M_{1j_1},..., M_{nj_n}\}).$$ ❑

The following theorem states that a sufficient condition for the compatibility of a set of actions diagrams is that the conjunction of their commit constraints be consistent and satisfies the assume constraints.

Theorem 1 [***Compatibility theorem***] Consider a set $Q = \{AD_1,..., AD_n\}$ of causal action diagrams where $AD_i = (S_i, \mathcal{A}_i, o_i, C_i)$, $i = 1,..., n$. Let *PConG* be a sound port connection group over Q, *ActConG* be the action connection group derived from $(PConG, \{o_1,..., o_n\})$ and $\mathcal{A}$ the set of communication actions of *ActConG*. Let C_{iC} (respectively C_{iA}) be the set of commit constraints (respectively assume constraints) of AD_i transposed to $\mathcal{A}$. Let $C_C = C_{1C} \cup ... \cup C_{nC}$, and $C_A = C_{1A} \cup ... \cup C_{nA}$. Then, if $Zone(\mathcal{A}, C_C)$ is non-empty and $Zone(\mathcal{A}, C_C) \subseteq Zone(\mathcal{A}, C_A)$, it follows that $AD_1,..., AD_n$ are compatible with respect to *PConG*.

Proof. Since $AD_1,..., AD_n$, are causal action diagrams, they each have at least one causal derived block machine. Let M_i be an arbitrary causal block machine of AD_i, $i = 1,..., n$, and let M be the composed block machine $Composed\text{-}BM(Con_a, \{M_1,..., M_n\})$. Since, by assumption, $Zone(\mathcal{A}, C_C)$ is non-empty, it follows that M is defined. Using Definition 46 and Definition 45, we must prove that M is causal and that all its traces are in $Zone(\mathcal{A}, C_A)$, i.e., satisfy the assume constraints.

Referring in sequential order to the three causality conditions of Definition 26, we prove in the following that M is causal:

1. By construction of the composed block machine (Definition 44), both the action direction condition (i.e., all actions local to a block have the same direction) and the constraint intent condition (i.e., all constraints local to a block have the same intent) are true.

2. Proof of *WDT*(M): Let B be a block of M, a_t a trigger of B, and a_l a local action of B. By construction, B is the transposition in M of an output block of M_i, for some i. Let a_t^i and a_l^i be the actions of M_i corresponding to a_t and a_l, respectively. Since M_i is causal, it satisfies the well-defined triggers condition, and thus $dist_{[M_i]}(a_l^i, a_t^i) < 0$. In addition, from Lemma 16, we have $dist_{[M]}(a_l, a_t) \leq dist_{[M_i]}(a_l^i, a_t^i)$. As a result, $dist_{[M]}(a_l, a_t) < 0$. It follows that M satisfies the well-defined triggers condition.

3. Proof of *past-dominated*(M): Let B be a block of M. If B has a single trigger, then *loose-blocks*(M) is trivially true, and hence, from Lemma 13, *past-dominated*(M) is also true. Otherwise, B has more than one trigger. Let (a_j, a_k) be a trigger pair of B, $a_j \neq a_k$. By construction, B is the transposition in M of an output block, say B_i, of M_i, for some i. Le: a_j^i and a_k^i be the actions of M_i corresponding to a_j and a_k, respectively. From Lemma 16, we have:

$$dist_{[M]}(a_l, a_t) \leq dist_{[M_i]}(a_l^i, a_t^i) \tag{17}$$

Since M_i is causal, *past-dominated*(M_i) is true. From Lemma 13, it follows that *loose-blocks*(M_i) is true, and thus:

$$dist_{[M_i]}(a_l^i, a_t^i) < dist_{[B_i]}(a_l^i, a_t^i) \tag{18}$$

Since B is the transposition of B_i in M, we obviously have:

$$dist_{[B_i]}(a_l^i, a_t^i) = dist_{[B]}(a_l, a_t) \tag{19}$$

From (17), (18), and (19), we get:

$$dist_{[M]}(a_l, a_t) < dist_{[B]}(a_l, a_t) \tag{20}$$

(20) implies that *loose-blocks*(M) is true, and hence, from Lemma 13, *past-dominated*(M) is true.

From items 1 to 3 above, it follows that M is causal. In the following, we prove that all the traces of M are in $Zone(\mathcal{A}, C_A)$, i.e., satisfy the assume constraints of

AD_1,..., AD_n. Let AD be the composed action diagram *Composed-AD*(*PCon*, $\{AD_1, ..., AD_n\}$). By assumption, AD is consistent (since *Zone*($\mathcal{A}$, C_C) is non-empty). Furthermore, the composed block machine M defined above satisfies the properties *WDT(M)* and *past-dominated(M)*, as proven above. Finally, $M \in dBMs(AD)$ is true, i.e., M is a block machine derived from AD; indeed, this property trivially follows from the construction, as given in Definition 44, of the composed block machine M and from the property *WDT(M)*. Hence, using Lemma 15, it follows that *TraceSet(M)* = *TraceSet(AD)*. Since *TraceSet(AD)* is given by *Zone*($\mathcal{A}$, C_C) and, by assumption, *Zone*($\mathcal{A}$, C_C) $\subseteq$ *Zone*($\mathcal{A}$, C_A), it follows that *TraceSet(M)* $\subseteq$ *Zone*($\mathcal{A}$, C_A), and hence *TraceSet(M)* satisfies the set C_A of assume constraints of AD_1,..., AD_n. ❑

Theorem 1 provides operational means for verifying the compatibility of causal interface specifications and thus the compatibility of any of their block machine based implementations. It suffices to verify that the maximum time distances between actions as determined by the composed system of commit constraints C_C, are contained in the time distances required by the assume constraints. In other words, the simple composition that was discussed in the example related to Figure 3 (in Section 3) is correct provided that the participating action diagrams are causal. This is clearly not the case for AD_2 of Figure 3, since the output block containing (necessarily) the only output action o4 does not satisfy the *loose-blocks* criterion (Definition 32) - the time distance between actions i2 and i3 using the local commit constraints is [-9, 9], while the time distance of its triggers as determined by all the constraints is also [-9, 9]. This interval is not *strictly* included in the former interval, hence the block machine is not causal. Since there is no other possible partition that satisfies the *loose-blocks* condition, the action diagram itself is not causal. Consequently, the compatibility check done by composing the commit constraints of the two action diagrams produced a false positive answer.

The composition of Figure 5 and Figure 6, as shown in Figure 7, satisfies all assume constraints, and since both action diagrams are causal, the compatibility decision is definitive.

12 Independence of Input and Output Sub-Partitions

In this section, we prove that the structure of the partition of the set of input actions of a causal block machine is independent of that of its output actions. In

other words, given two causal block machines derived from the same action diagram, then the block machine derived by "cross-breeding" the input action sub-partition of one of the machines with the output action sub-partition of the other machine, is also a causal block machine. This property, which comes about as a corollary of Theorem 1, is intuitively "reassuring" and is one more indication of the "soundness" of the work presented in this paper. The property should also be useful in designing an efficient partitioning procedure.

Definition 47 [*In/out spec of a block machine*] Let $M = (\mathcal{A}, o, \mathcal{B}, \mathcal{T}, C)$ be a derived block machine such that the local actions of any block of M are of the same direction. The input spec, IS, (respectively output spec, OS) of M is the tuple $(\mathcal{A}', o, \mathcal{B}', \mathcal{T}', C')$, where $\mathcal{A}'$ is the subset of input (respectively output) actions of $\mathcal{A}$, $\mathcal{B}'$ is the subset of input (respectively output) blocks of $\mathcal{B}$, $\mathcal{T}'$ is the restriction of $\mathcal{T}$ to $\mathcal{B}'$, and C' is the subset of *assume* (respectively *commit*) constraints of C. Clearly, M is uniquely defined by the pair IS, OS. We write: $M \approx (IS, OS)$. ❑

Definition 48 [*Mirror of a block machine*] Let $M \approx (IS, OS)$ be a derived block machine, such that the local actions of any block of M are of the same direction. The mirror of IS (respectively OS) is the output spec $\overline{IS}$ (respectively input spec $\overline{OS}$) obtained from IS (respectively OS) by changing the direction associated with every *in* (respectively *out*) action of IS (respectively OS) to *out* (respectively *in*) and changing the intent associated with every constraint of IS (respectively OS) to *commit* (respectively *assume*). The mirror of M is the block machine $\overline{M}$, where $\overline{M} \approx (\overline{OS}, \overline{IS})$. ❑

Obviously, mirroring a block machine M does not change its global zone, since M and $\overline{M}$ have the same structure and weights of constraints. Hence, if Z (respectively $\overline{Z}$) is the global zone of block machine M (respectively $\overline{M}$), then $Z = \overline{Z}$.

Corollary 1 [*Independence of input and output specs*] Let AD be a causal action diagram and M_1 and M_2 two causal block machines derived from AD, such that $M_1 \approx (IS_1, OS_1)$ and $M_2 \approx (IS_2, OS_2)$. Then, the block machines $M_{12} \approx (IS_1, OS_2)$ and $M_{21} \approx (IS_2, OS_1)$ are also causal.

Proof. Let $\overline{M_1}$ be the mirror of M_1, i.e., $\overline{M_1} \approx (\overline{OS_1}, \overline{IS_1})$, $Q = \{\overline{M_1}, M_2\}$ a set of block machines, *Con* a sound action connection group over Q such that the action set of each action connection of Q is of the form $\{a_i, a_i\}$. Consider the composed machine M_c = *Composed-BM*(*Con*, $\{M_1, M_2\}$). By construction, $M_c \approx (IS_1, OS_2)$. Furthermore, M_c satisfies the well-defined triggers conditions, i.e., $WDT(M_c)$ is true (the proof is the same as that for the statement $WDT(M)$ in the proof of Theorem 1). Hence, using the fact that the constraint set of IS_1 is the set of assume constraints of *AD*, and the constraint set of OS_2 is the set of commit constraints of *AD*, it follows that Z_c, the global zone of M_c, is the same as Z_{AD}, the global zone of *AD*, i.e., $Z_c = Z_{AD}$. This implies that Z_c is non-empty (since *AD* is consistent) and that Z_c satisfies the commit and assume constraints of *AD*, respectively. Hence Z_c satisfies the assume constraints of M_1 and the assume constraints of M_2, respectively. From Theorem 1, it follows that M_1 and M_2 are compatible for the action connection set *Con*, and hence, from Definition 46, it follows that M_c is causal. Now since $M_c \approx (IS_1, OS_2)$, and IS_1 is identical to IS_1, it follows that $M_{12} \approx (IS_1, OS_2)$ is causal too. The proof is symmetrical for M_{21}. ❑

13 Conclusion

We have defined sufficient conditions for a specification based on action diagrams with linear timing constraints to be causal and thus realizable, and we have developed a method for determining the causality of an action diagram. This has lead to a procedure for verifying the interface compatibility of communicating action diagrams. The results are useful for writing action diagram specifications, verifying interoperability of systems composed of communicating components, and for implementing interface controllers. Our causality criterion is considerably more general than the *well-posedness* criterion of [Ku92]. The latter is in effect equivalent to requiring that all actions of the same direction (*in* or *out*) be in the same block. In addition, well-posedness does not take into account timing assumptions on the environment; instead, it requires that the device responds to arbitrary timing behaviors of the environment.

We are currently researching algorithms for the efficient determination of action partitions that yield causal block machines. We are also working on extending our approach to cyclic behaviors. A natural extension of the approach is to include the *latest* constraints [Amon93] in addition to the linear

constraints. They are by their nature causal, and efficient methods exist for computing the shortest distances over linear and latest constraint systems [MacM92, Giro95]. The inclusion of *earliest* constraint makes the problem of computing time distances between actions NP-complete [MacM92], however, as shown in [Giro95], we can use CLP (BNR) Prolog and its power of relational interval arithmetic to solve the constraint satisfaction problem and to perform the necessary exploration and backtracking.

References

[Amon93] T. Amon, H. Hulgaard, G. Borriello, S. Burns, "Timing Analysis of Concurrent Systems: An Algorithm for Determining Time Separation of Events", *Proc. ICCD-93*, October 1993.

[Borr88] G. Borriello, "A New Interface Specification Methodology and its Application to Transducer Synthesis", *Ph.D. Thesis*, EECS, University of California, Berkeley, 1988.

[Brzo91] J.A. Brzozowski, T. Gahlinger and F. Mavaddat, "Consistency and Satisfiability of Waveform Timing Specifications", *Networks*, Vol. 21, 1991, pp91-107.

[Burk93] T.M. Burks and K.A. Sakallah, "Min-Max Linear Programming and the Timing Analysis of Digital Circuits", *Proc. ICCD-93*, October 1993, pp152-155.

[CCITT] Recommendation Z.120, CCITT. "Message Sequence Charts (MSC)".

[Dill89] D. Dill, "Timing Assumptions and Verification of Finite State Concurrent Systems", *International Workshop on the Verification of Finite State Systems*, Grenoble France, 1989. Also in *Lecture Notes in Computer Science (LNCS) 407, Springer Verlag*, 1989.

[Giro95] P. Girodias, E. Cerny, W.J. Older, "Solving Linear, Min and Max Constraint Systems Using CLP Based on Relational Arithmetic," submitted to *Int'l Conf. on Principles and Practice of Constraint Programming (CP95)*, Marseille, September 1995.

[Hulg93] H. Hulgaard, S.M. Burns, T. Amon and G. Borriello, "Practical applications of an efficient time separation of events algorithm", *Proc. ICCAD-93*, Santa Clara, CA, November 1993.

[Khor93] K. Khordoc, M. Dufresne, E. Cerny, P.-A. Babkine and Allan Silburt, "Integrating Behavior and Timing in Executable Specifications", *Proc. CHDL'93*, April 1993.
[Khor94] K. Khordoc and E. Cerny, "Modeling Cell-Processing Hardware with Action Diagrams", *Proc. ISCAS-94*, June 1994.

[Klus93] A.S. Klusner, "Models and axioms for a fragment of real time process algebra", *Ph.D thesis*, CWI, Amsterdam, 1993.

[Ku92] D. C. Ku and G. De Micheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*, Kluwer Academic Publishers, 1992.

[MacM92] K. McMillan and D. Dill, "Algorithms for Interface Timing Verification, *Proc. ICCD-92*, October 1992.

[Rony80] P. Rony, "Interfacing Fundamentals: Timing Diagram Conventions", *Computer Design*, January 1980, pp152-153.

[Tarj83] R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM 1983.

[Wiat80] C. Wiatrowski and C. House, Logic Circuits and Microcomputer Systems, *McGraw-Hill*, New York, 1980.

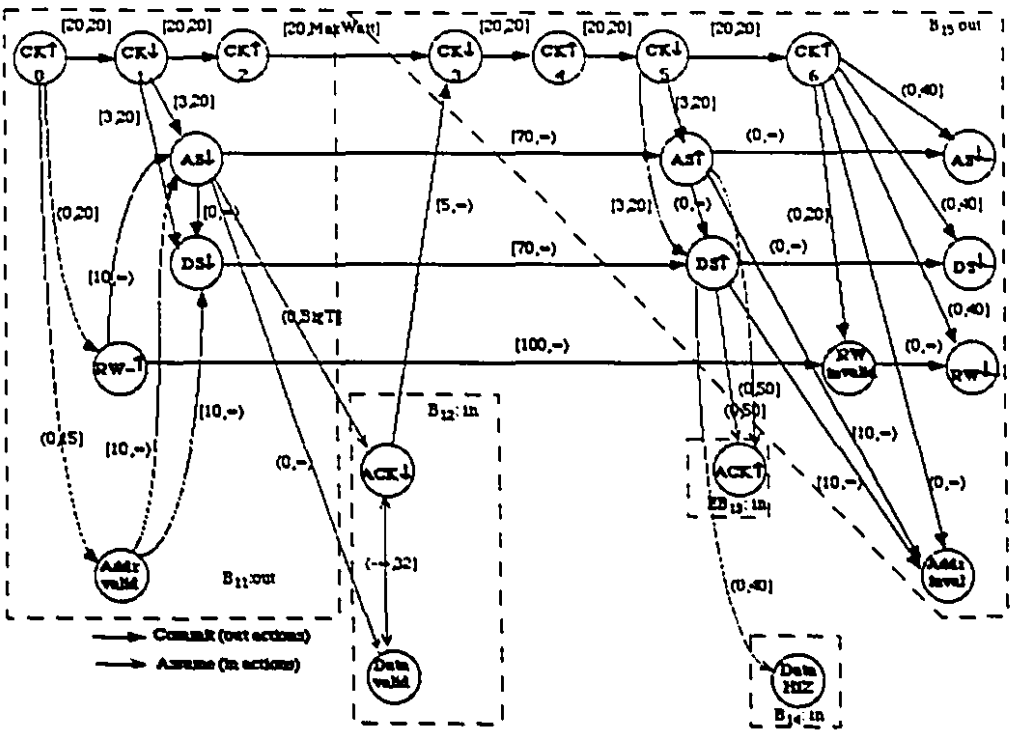


Figure 5: Action Diagram for the READ cycle of the MC68360 processor (wait states are modeled by delay interval between CK2 and CK3).

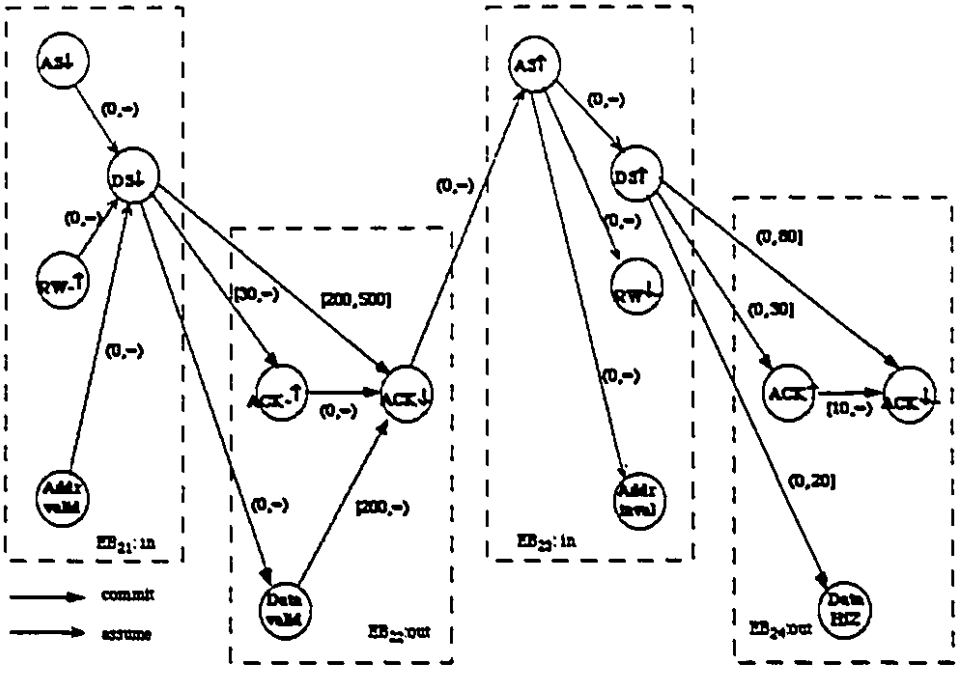


Figure 6: Action Diagram for the READ cycle of the slave device.

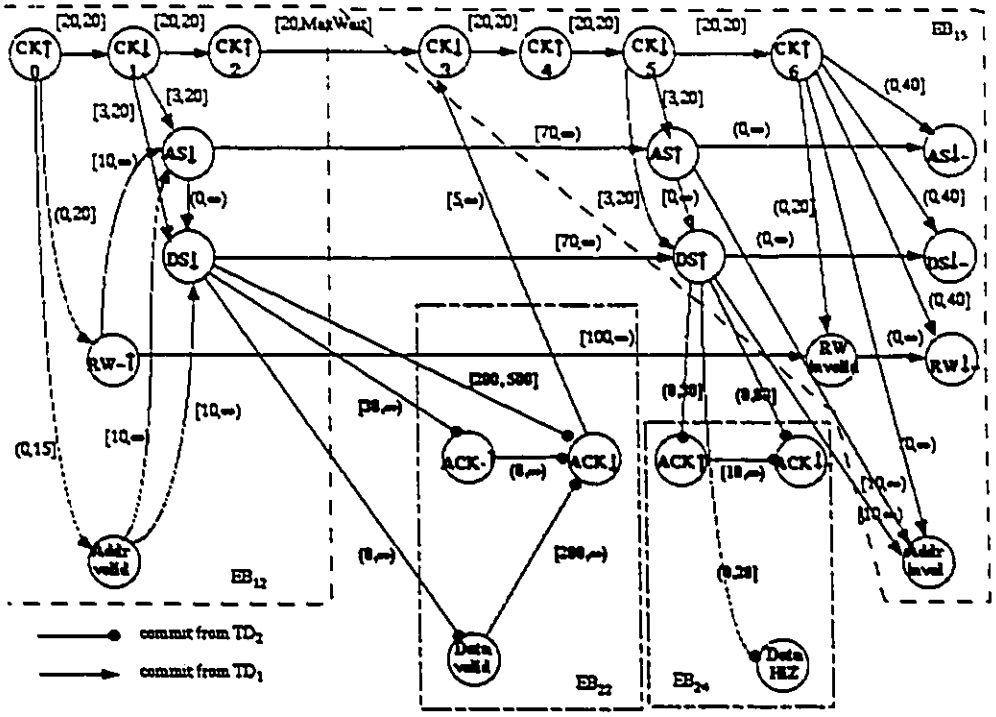


Figure 7: Composition P_{12} resulting from Figure 5 and Figure 6.

CHAPTER 7

GENERAL CONCLUSIONS

1 Summary

In this thesis, we addressed issues in the specification, simulation, and formal verification of systems that are characterized by real-time requirements and a mix of protocol and data computation aspects.

We proposed the HAAD (Hierarchical Annotated Action Diagrams) specification language and modeling methodology. In HAAD, the interface behavior is captured separately from the internal behavior while maintaining the links between the two. The interface behavior is captured as a hierarchy of action diagrams. The internal behavior is modeled by an Extended Finite State Machine (EFSM). We proposed to link the interface behavior and internal behavior by shared variables and *synchronization points*. A leaf action diagram defines a behavior (a template) over a set of ports. The behavior of a port is captured as a sequence of actions (events). Actions can be related by min. / max. weighted timing constraints which capture precedence, concurrency and causality relations between the actions. The constraints describe the assumptions that the behavior makes on its environment as well as the way in which the behavior reacts to its environment. The functional description of the system interface is included in a HAAD specification by defining state variables, input/output parameters, and by attaching procedures and predicates to actions. Hierarchical action diagrams are constructed by composing other action diagrams (leaf or composed) using the composition operators: *Concatenation*, *Loop*, *Concurrency*, *Choice*, and *Exception Handling*. The Choice semantics support three specification styles that we found to be useful at the system level. The Choice can be deterministic, delayed-deterministic, or non-deterministic. The delayed-deterministic semantics allow system spec-

ifications to be given in a scenario-based style. The non-deterministic style supports design abstractions.

We proposed algorithms and methods for the automatic generation of simulation models and response verification scripts from HAAD specifications. These models perform "on-the-fly parsing" of events received at their I/O ports, sequencing through state transitions based on the result of this parsing, detecting incorrect, or ill-formed interface operations (bus cycles), verifying that all timing constraints at the input of the model are met, and driving the model outputs with appropriate delays.

We formalized the operational semantics of leaf action diagrams under linear timing constraints, based on the concepts of a *block machine* and *causal* block machine. We stated the realizability of an action diagram in terms of the existence of a causal block machine derived from the action diagram. We examined the problem of the compatibility of concurrent, communicating leaf action diagrams described by linear timing constraints and we showed the inaccuracies of known methods that address this problem. We defined the action diagram compatibility problem in terms of the compatibility of *all* the possible combinations of causal block machines derived from these action diagrams. We proved that such enumeration is not needed in answering the compatibility question. This lead to an exact and efficient compatibility verification procedure.

2 Benefits of our Work

The benefits of our work are summarized in the following:

- Our proposed modeling methodology, HAAD - Hierarchical Annotated Action Diagrams - facilitates the modeling of real-time systems.
- The structure of HAAD models is more amenable to automated analysis.
- The HAAD focus on high-level formal specifications of sub-system interfaces early in the design cycle, coupled with the natural declarative style of action diagrams decreases the chances of interface mismatches at system integration time.
- Our capability in automatic executable model generation markedly reduces the time that designers spend writing test benches.

- Our well-behavedness criteria and compatibility analysis of timing diagrams help improve the quality of interface designs and minimize the time spent in costly design reworks.

3 Original Contributions

1- The original contributions of the HAAD modeling language and methodology are:

- the separation of, and links between, interface behavior and internal behavior,
- the separation of, and links between, functional aspects and protocol/timing aspects in interface specifications,
- the combination of a true behavioral hierarchy *and* a rich set of timing constructs, and
- the delayed choice semantics.

2- In the area of executable model generation, our original contributions are:

- a novel algorithm for dynamic stimulus generation and response validation from timing diagrams,
- a unified framework for *valid* and *don't care* signal states, and
- a unified approach to model generation for master, slave and mixed behaviors.

3- Our original contributions in formal timing verification are:

- sufficient conditions for the well-behavedness of interface specifications under linear timing constraints,
- operational semantics of interface specifications under linear timing constraints,
- analysis of false negatives and false positives in known compatibility verification methods, and
- an accurate compatibility verification procedure for timing diagrams.

4 Recommendations for Further Research

Relaxing the strict encapsulation of action diagrams

The strict encapsulation of behaviors into action diagrams using Start/End pseudo-actions (Chapters 4 and 5) is elegant and facilitates both the simulation and formal analysis of the specifications. However, it is sometimes intuitively sound from a modeling point of view to express partially overlapping interface operations, i.e., that an interface operation be activated while there are still some ("tail-end") actions that have not yet occurred in the previously executing interface operation. In the present HAAD framework, such a situation cannot be directly modeled. Instead, the specification may need to be partitioned into individual action diagrams along non-intuitive boundaries (rather than the natural boundaries between interface operations). This requires some modeling effort and the resulting model is generally more difficult to understand. Hence, additional work is needed to explore the relaxation of action diagram encapsulation and allow partially overlapping interface operations.

Expressing pipelined behaviors

Perhaps a more general problem than that of overlapping interface operations is that of pipelined behaviors. From a modeling point of view, it often is desirable to capture in one leaf action diagram the cause-to-effect relationship and delay (i.e., pipeline latency) from an input action of the pipeline to its logically related output action. This cannot be done in the present HAAD framework. Instead, in the case of a constant rate pipeline, i.e., with inputs (outputs) arriving (departing) at a constant rate, the main behavior loop of the model would be around a leaf action diagram containing unrelated (function wise) input and output actions. As for variable rate pipelines, they cannot be modeled in the present HAAD framework. Thus, additional work is needed in the area of pipeline modeling. One possibility is to define a pipelining operator.

Inter-diagram timing constraints

In many bus interface specifications, there are timing constraints between actions of an interface operation and the next. Such constraints cannot be expressed in the present HAAD framework. Instead, the user resorts to either a less accurate timing model based on timing constraints relative to the Start/End actions of the action diagrams, or redefines the inter-diagram boundaries so that no timing constraint crosses them (which

often results in unnatural models). More work is needed to explore the repercussions of inter-diagram timing constraints on the semantics and analysis algorithms associated with HAAD.

Unifying the timing constraint model

In the present version of HAAD simulation tools, commit constraints are restricted to non-linear and assume constraints can be linear or non-linear. As for the formal compatibility verification tool, both commit and assume constraints are restricted to be linear. More work is needed in generalizing the block machine model to include the non-linear constraints in a unified semantic framework. This framework must then be the basis of both simulation and formal verification.

Relaxing the causality criterion

In the definition of block machines (Chapter 6), the selection of trigger actions is syntactic, i.e., it is affected by, amongst other things, the structure of the constraint system. Furthermore, the block machine semantics require a strict trigger concept, i.e., all the triggers of a block must occur before the block is enabled. These two restrictions could rule out valid implementations of a specification. Consider, for example, the non-causal specification of Figure 1, Chapter 6. As proposed by [1], a valid implementation *does* exist for this specification. By examining this implementation as well as a family of similar implementations, we realized that these could be arrived at by generalizing the trigger concept to be non-syntactic (i.e., not apparent from the structure of the constraint system) and non-strict (i.e., using the *earliest* operator). Hence, more work is needed in exploring the generalization of block machines and the possible relaxation of the causality criterion.

Causal machine derivation

Chapter 6 dealt with the derivation of a block machine, given a partition of the action set of the timing diagram. More work needs to be done to develop algorithms and heuristics for the derivation of the actual action partition that defines a causal block machine.

Formal verification of HAAD specifications

In Chapter 6 we developed an efficient static[1] compatibility verifi-

[1]The verification procedure is static in the sense that it does not perform any state space exploration.

cation procedure for leaf action diagrams. It is interesting to note that two related projects undertaken by our colleagues at LASSO [2] are relevant to the continuation of our work. The first project [2] reports on the formal verification of general (non-annotated) HAAD specifications. The approach is general, however it is limited in efficiency due to its reliance on full interleaving. The second project [3] uses CLP-BNR [4], a general-purpose computational environment based on constraint logic programming (CLP) and relational interval arithmetics (RIA), to solve the maximal time distance problem for mixed linear and non-linear constraints, while taking into account the effects of delay correlation. It appears possible to combine the strengths of the three approaches, i.e., analyze general (non-annotated) HAAD specifications using an overall state-space exploration approach combined with a static analysis at the leaf level. The static analysis would be based on the approach described in Chapter 6 and implemented in a CLP/IRA environment. Evidently, there is more work that needs to be done to make this possible.

References

[1] M. Aboulhamid, Professor, département d'informatique, Université de Montréal, *private communication.*

[2] B. Berkane, S. Gandrabur, and E. Cerny, "Timing diagrams: semantics and timing analysis", *Proceedings of the Asian Pacific Conference on Computer Hardware Description Languages*, 1996.

[3] P. Girodias, E. Cerny, W.J. Older, "Solving Linear, Min and Max Constraint Systems Using CLP Based on Relational Arithmetic," submitted to Int'l Conf. on Principles and Practice of Constraint Programming (CP95), Marseille, September 1995.

[4] W. Older and A. Vellino, "Constraint Arithmetic on Real Intervals", *Constraints Logic Programming: Selected Research*, 1993.

[2] Laboratoire d'Analyse et de Synthèse des Systèmes Ordinés, département d'informatique, Université de Montréal.

APPENDIX I

SYNTACTIC WELL-FORMEDNESS RULES FOR ACTION DIAGRAMS

1 Introduction

This appendix lists syntactic well-formedness rules for action diagrams. The rules of Section 2 reflect the simplifying design decisions that were made in the the present version of the HAAD simulation tools. These rules can be relaxed by integrating the causality framework of Chapter 6 into the HAAD simulation engine. The rules of Section 3 establish the restrictions under which the algorithms of Chapter 3 behave meaningfully.

2 Strict Causality in HAAD Simulation

- Every *output* and *internal* action must be the sink of at least one *commit* constraint.
- All commits constraints must be:
  - of type *precedence*
  - bounded (i.e., finite u in $[l, u]$)
  - composed only with the *Earliest* or *Latest* operators (no conjunctive composition).

The advantages of this "strict causal style" are twofold: 1- the causality information (i.e., what actions cause what other output or internal

actions) is explicit, and 2- model interpretation (simulation) is efficient, using a relatively simple algorithm. Note however that in general, writing specifications in this causal style requires more information on the modeled system.

3 Assume Constraints and Input Don't Care Events

- An input *Don't Care* event cannot be the source of a timing constraint (whether assume, or commit).
- The event following an input *Don't Care* event on the same port, cannot be the source of a commit constraint.
- Assume constraints that have an input action sink of spec value *Don't Care* are half-bounded min only (i.e., $u = +\infty$ in $[l, u]$) precedence constraints from actions of *constant* spec value (e.g., E_1 in Fig. 1) to the input *Don't Care* action.
- Constraints related to the input action (say *Next*) that follows an input *Don't Care* action on the same port can be:
  - Half-bounded max only precedence assume constraints from actions of *constant* spec value (e.g., E_2 in Fig. 1) to *Next*.
  - Half-bounded min precedence assume constraints from *Next* to actions of *constant* spec value (e.g., E_3 in Fig. 1).
  - At least one of the two above situations must be true of *Next*.

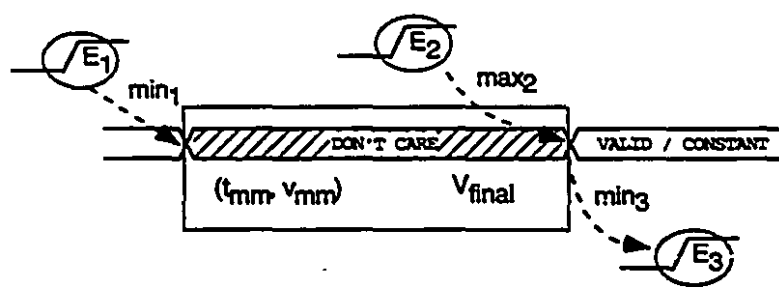


Figure 1: Allowed constraints on input *don't care* and *valid* actions.

APPENDIX II

THE DEFBEHAVIOR LANGUAGE

1 Introduction

A HAAD specification is captured with the *defbehavior* language. This appendix is the definition of the grammar of that language, i.e., it is an implicit definition of the set of sentences that form the language (from a syntactic point of view, a language is simply a set of sentences). This does not mean that all sentences of the language have associated semantics. The appendix contains "semantic notes" (Section 4), that are helpful in bridging the gap form the "set of sentences" view to the real semantics. More work is needed to complete this documentation.

The *defbehavior* language syntax follows a style that we designate as "Keyed List Language" (KLL). The KLL concept (Section 2) is inspired by the EDIF [1] language.

2 Keyed List Languages

Consider first a syntactic class of languages designated as "List Languages" (LL's):

- A sentence in a LL is a *list*.
- A list is syntactically delimited by a pair of parentheses.
- Each element of a list is an *atom* or a *list*.

- For our purposes, it suffices to define 3 types of atoms: symbol, number, and string.
- For the lexical rules (i.e., what ASCII character sequences make up symbols, numbers and strings, comment syntax, delimiters etc), we adopted the lexical rules of [2].

Then, consider a subclass of list languages, denoted "Keyed List Languages" (KLL's):

- A KLL is characterized by a set of *keys*, i.e., pre-determined symbols.
- In a KLL, all lists are "keyed", i.e., the first element of each non-empty list is a *key*.
- A keyed list is said to be a "form".
- The defbehavior language defined in this document is a KLL.

  In the following, and in order not to confuse the concept of a "grammar symbol" (i.e., terminal and non-terminal symbols used in the grammar that defines a language) with that of a "Lisp symbol", we use the terminology "item" for the former and "symbol" for the latter.

3 Conventions used in the Definition of the Defbehavior Grammar

- Note: The defbehavior language is case insensitive
- The grammar of the defbehavior language is specified in EBNF (Extended Backus-Naur form).
- In this EBNF, an upper case item indicates a terminal constant.
- Each lower case item is one of the following:
  - A non-terminal: these are those items that appear at the left hand side of EBNF productions.
  - a general lisp expression: the only such item is "lisp-expression" (See the last production in the grammar definition of Section 5). This is for future extensions of the language.

  - A general terminal: except for the item "lisp-expression", these are all lower case items that do not appear in any left hand side of EBNF productions. In terms of the EBNF, There are 3 types of general terminals (i.e., automatically recognized as "typed tokens" by lexical analysis): number, symbol, and string.

- A choice is indicated with a vertical bar. Only one of the options may be chosen.

- A list of 1 or more items enclosed within curly braces and separated by vertical bars (in the case of a list of length greater than one) indicates that any number of each item may be present and that the items may occur in any order. Inside such a list, if an item is permitted to occur at most once, it is enclosed within chevrons.

- In the grammar specification, we use convenient mnemonic names for these general terminals depending on their role in a construct.

- The general terminals of type "number" are:

  - number

- The general terminals of type "symbol" are:

  - had-type-nameDef
  - had-instance-nameDef
  - port-nameDef
  - signal-nameDef
  - param-nameDef
  - var-nameDef
  - generic-nameDef
  - action-nameDef
  - tc-nameDef
  - had-type-nameRef
  - var-nameRef
  - var-or-param-nameRef
  - signal-or-port-nameRef
  - var-or-param-or-signal-or-port-nameRef
  - source-action-nameRef

  - sink-action-nameRef

- General terminals of the symbol type can be quoted (i.e., preceded by the single quote character) or not. The language supports both. However, for backward compatibility with previous implementations of the defbehavior parser, the following symbols must be quoted):

  - action-nameDef
  - source-action-nameRef
  - sink-action-nameRef
  - tc-nameDef
  - had-instance-nameDef
  - had-type-nameRef

- The general terminals of type "string" are:

  - v-prog-nameRef
  - v-type-nameRef
  - v-value

4 Semantic Notes

4.1 Generics

- The only lower case item of the grammar that does not appear in any left hand side of a EBNF production is the item "lisp-expression".

- The item "lisp-expression" (which appears only in the right-hand side of the EBNF production for the "generic-map" item), stands for a general lisp expression. This Lisp expression is evaluated at design instantiation time in the lexical scope of the *current* defbehavior (i.e., the one containing the generic-map form). In the *instantiated* defbehavior (i.e., the one that is instantiated as a sub-behavior of the current defbehavior), all occurrences of the generic to which this lisp-expression was mapped to, are replaced by the value of this lisp-expression.

4.2 Default Constraint Bounds

The semantic interpretation of constraint bounds, in the absence of min-spec and/or max-spec sub-forms in the PRECEDENCE and/or CONCURRENCY forms is:

- if no min-spec is specified in a PRECEDENCE form, it is semantically equivalent to a strict lower bound of 0.
- if no min-spec is specified in a CONCURRENCY form, it is semantically equivalent to no lower bound specification (i.e., a minus infinity lower bound).
- if no max-spec is specified in a PRECEDENCE or CONCURRENCY form, it is semantically equivalent to no upper bound specification (i.e., a plus infinity upper bound).

5 Grammar Definition

```
defbehavior ::= (DEFBEHAVIOR had-type-nameDef
                                  { <ports> | <parameters> |
                                    <generics> |
                                    signal | var |
                                    <had-body> })


ports ::= (PORTS { port })

port ::= (PORT port-nameDef direction v-type-nameRef
                                        interpretation)

direction ::= INOUT | IN | OUT


interpretation ::= EVENT | MESSAGE

parameters ::= (PARAMS { parameter })

parameter ::= (PARAM param-nameDef direction v-type-nameRef)
```

```
generics ::= (GENERICS { generic-nameDef })

signal ::= (SIGNAL signal-nameDef v-type-nameRef
                                    interpretation)


var ::= (VAR var-nameDef v-type-nameRef { <v-value> })


had-body ::= leaf |
         had-loop |
         concatenation |
         parallel |
         d-choice |
         nd-choice |
         exception

leaf ::= (LEAF { carrier-spec | constraint |
                 <start-action> | <end-action> })

carrier-spec ::= (CARRIER-SPEC signal-or-port-nameRef
                                { <initial-spec> |
                                     action-spec } )



initial-spec ::=  (INITIAL-SPEC state {<action-direction>})

action-direction ::= IN | OUT

action-spec ::= (ACTION-SPEC action-nameDef state {
                                    <action-direction-spec> |
                                    <predicate-call>    |
                                    <procedure-call>    } )

action-direction-spec :: (DIRECTION action-direction)


state ::= dont-care | constant | valid
```

```
dont-care ::=  (DONT-CARE)

constant ::=  (CONSTANT v-value)

valid ::= (VALID { <var-nameRef>  })


procedure-call ::= (PROCEDURE-CALL v-prog-nameRef
                  {var-or-param-or-signal-or-port-nameRef})

predicate-call ::= (PREDICATE-CALL v-prog-nameRef
                  { var-or-param-or-signal-or-port-nameRef})

constraint ::= conjunctive | earliest | latest |
               precedence | concurrency

conjunctive::= (CONJUNCTIVE { <tc-name-spec> | constraint } )

earliest   ::= (EARLIEST    { <tc-name-spec> | constraint } )

latest     ::= (LATEST      { <tc-name-spec> | constraint } )


precedence  ::= (PRECEDENCE source-action-nameRef
                            sink-action-nameRef
                          { <tc-name-spec> | <intent-spec> |
                              <min-spec> | <max-spec> })

concurrency ::= (CONCURRENCY source-action-nameRef
                             sink-action-nameRef
                           { <tc-name-spec> | <intent-spec> |
                               <min-spec> | <max-spec> })

tc-name-spec ::= (CNAME tc-nameDef)


intent-spec ::= (INTENT intent)

intent ::= ASSUME | COMMIT | REQUIREMENT

min-spec ::= (CMIN min)
```

```
max-spec ::= (CMAX max)


min ::= number

max ::= number


had-loop ::= (HAD-LOOP had
              { <start-action> | <predicate-call> |
                <end-action> } )


concatenation ::= (CONCATENATION
                   { <start-action> | had | <end-action> })

parallel ::= (PARALLEL
                   { <start-action> | had | <end-action> })


d-choice ::=  (D-CHOICE
               { <start-action> | choice-branch |
                                  <end-action> } )

nd-choice ::= (ND-CHOICE
               { <start-action> | choice-branch |
                                   <end-action> } )


choice-branch ::= (BRANCH had { <predicate-call> } )

start-action ::= (START-ACTION
                   { <predicate-call> | <procedure-call> })

end-action ::= (END-ACTION
                   { <predicate-call> | <procedure-call> })

exception ::= (EXCEPTION {<condition> | <normal> | <handler>
                          | <start-action> | <end-action>})


condition ::= (CONDITION had)
```

```
normal ::= (NORMAL had)

handler ::= (HANDLER had)

had ::= had-instance-spec | had-body

had-instance-spec ::= (BEHAVIOR had-instance-nameDef
                          had-type-nameRef
                        { <port-map> | <param-map> |
                          <generic-map> } )

port-map ::= (PORT-MAP { signal-or-port-nameRef })

param-map ::= (PARAM-MAP { var-or-param-nameRef })


generic-map ::= (GENERIC-MAP { lisp-expression })
```

References

[1] "EDIF - Electronic Design Interchange Format, Version 2.0.0", *Electronics Industries Association*, 1987.

[2] "The Common Lisp Language", X3J13, ANSI X3.226:1994, *American National Standard for Programming Language*, 1994.