# Formally Specifying and Verifying Real-Time Systems

Richard A. Kemmerer  Paul Z. Kolano
Reliable Software Group
Computer Science Department
University of California
Santa Barbara, CA 93106 USA
+1 805 893 4232
{kemm,kolano}@cs.ucsb.edu

**ABSTRACT**

*A real-time computer system is a system that must perform its functions within specified time bounds. These systems are generally characterized by complex interactions with the environment in which they operate and strict time constraints whose violation may have catastrophic consequences. The need for these software systems to be highly reliable is evident. One way to achieve this reliability is through formal development.*

*Although research in the area of real-time systems has been quite active and a number of experimental environments supporting formal specifications have been developed, the search for adequate notations and tools is still ongoing. In order to get designers to use formal methods to develop real-time systems it is necessary to provide them with an integrated set of tools for writing and analyzing their specifications. The ASTRAL Software Development Environment (SDE), which is an integrated set of tools based on the ASTRAL formal framework, is intended to meet this need. The tools that make up the support environment are a syntax-directed editor, a specification processor, a verification condition generator, a mechanical theorem prover, and a browser kit.*

*This paper discusses the goals for ASTRAL, why they were important, and how they were met. It will also give an overview of the ASTRAL Software Development Environment.*

## 1. INTRODUCTION

A real-time computer system is a system that must perform its functions within specified time bounds. Real-time computer systems are increasingly being used in critical applications such as aircraft avionics, nuclear power plant control and patient monitoring.These systems are generally characterized by complex interactions with the environments in which they operate, and strict time constraints whose violation may have catastrophic consequences. The need for these software systems to be highly reliable is evident.

More than two decades ago Wirth [Wir 77] classified programs into three types sequential, parallel, and "processing-time dependent" (real-time). The complexity of writing correct systems increases with the introduction of parallelism and is further complicated with the introduction of real-time constraints. Similarly, the difficulty of specification and verification increases from sequential to parallel to real-time systems.

Like sequential systems, real-time systems are judged against critical correctness requirements. That is, both sequential systems and real-time systems have critical functionality requirements. Real-time systems, however, must also meet critical performance deadlines. If a real-time system performs the correct function, but delivers the result too late, then it has failed to satisfy its requirements. For example, if a process monitoring a nuclear reactor detects a malfunction but does not respond in a timely fashion this would violate a performance criteria and likely jeopardize the safety of the system and endanger human life. Therefore, the verification of real-time systems involves demonstrating that the specified system meets the performance deadlines in every case and, in particular, in the worst case.

Some of the issues involved in building real-time systems are real-time architectures, operating systems, programming languages, scheduling algorithms and communication protocols. Many real-time systems have been built and apparently work. Nevertheless, today many such systems are being built using algorithms, techniques and tools that might be described as marginally adequate for sequential systems, less than adequate for concurrent systems, and wholly unacceptable for real-time systems in critical applications. Improvements are needed in several of the above mentioned areas, each of which represents an important research area in its own right, but the topic addressed in this paper is improving system reliability through better software development methods and tools, and specifically, through applying formal methods to the development of real-time systems.

Although research in the area of real-time systems has been quite active and a number of experimental environments supporting formal specifications have been developed, the search for adequate notations and tools is

still ongoing. In order to get designers to use formal methods to develop real-time systems it is necessary to provide them with an integrated set of tools for writing and analyzing their specifications. The ASTRAL Software Development Environment (SDE), which is an integrated set of tools based on the ASTRAL formal framework, is intended to meet this need.

The Reliable Software Group at UCSB has designed and implemented a language for formally specifying and verifying sequential software systems, called ASLAN [AK 85]. In addition, they have designed an extension of the ASLAN specification language called RT-ASLAN, for specifying real-time systems [AK 86]. The ASLAN specification language served as a basis for the ASTRAL language and some of the RT-ASLAN approaches were also adapted to ASTRAL. The ASTRAL language, however, was developed as a new language. Although the ASLAN state machine approach with layering is retained, ASTRAL uses a novel approach to modeling interprocess communication, and many new specification ideas are introduced for expressing interaction with the environment and timing relationships.

TRIO is a logic language designed at the Politecnico di Milano as a formal notation for specifying and verifying timing requirements [GMM 90]. The research and experimentation on TRIO initially addressed the issue of executing TRIO specifications [MMG 92]. Thus, TRIO can be considered as a real-time machine level formal language and this is why it was decided to build a high level language that could be translated to TRIO. The TRIO language was later extended with suitable object oriented mechanisms for modularizing a complex specification [MS 94]. However, it still lacks many useful concepts which are specific to ASTRAL, such as assumptions about the environment, critical requirements, and a modular proof system.

Using this ASLAN and TRIO experience, a new formal specification language for real-time systems has been developed (ASTRAL), and several case studies have been developed using this language. The semantics of the language have also been formally defined using both a model theoretic approach and an axiomatic semantic approach [CSK 94]. This provides a firm theoretical basis for the development of an ASTRAL support environment, which constitutes one of the on-going research directions. A translation scheme for translating ASTRAL to TRIO was defined for an earlier version of the ASTRAL language [GK 91]. The experience of basing the ASTRAL language on ASLAN and translating ASTRAL specifications into the TRIO logic language prompted the selection of the name for the language: an ASlan based TRio Assertion Language.

In the next section the goals and assumptions for this work are presented. This is followed by an introduction to the ASTRAL language. Next an overview of the ASTRAL Software Development Environment will be presented. Finally, conclusions drawn from this work are discussed.

## 2. GOALS AND ASSUMPTIONS

Because there is no point in developing another language that no developer would choose to use, usability was emphasized as a main goal in the design of ASTRAL. Whenever more than one language design choice existed, the option that made the language more usable was picked. Similarly, developing a language without state-of-the-art tools to support its use inevitably results in an unused language. Therefore, the development of ASTRAL proceeded in parallel with the initial design of tools for supporting the language. The plausibility of modifying and/or extending existing tools, such as the ASLAN specification processor or the TRIO Executor, for use with ASTRAL specifications was also constantly factored into the language design process, although this was not a primary factor in the decision making process.

In addition to having a language that is usable with tool support, the other primary goals for the ASTRAL specification language were that it support specifications that are layered, compositional, and executable. Layering and composition are two complementary approaches to hierarchical system development. A layered specification method allows one to refine the specification of a process to show more detail, without changing the interface of the specified system. This is important because it allows designers to prove, test, or otherwise examine properties of a process whose behavior is specified abstractly, and then iteratively refine the behavioral specification to be as close to an implementation as appropriate for a given assurance level. In this way errors can be found early in the design process, before spending time and money adding details.

A compositional specification method allows one to reason about the behavior of a system in terms of the specifications of its components [Zwi 89]. That is, the behavior of a system comprising several component processes is completely determined by the component specifications. This is important because it modularizes a system's proof and allows for bottom-up development.

An executable specification language allows developers to treat specifications as prototypes. This is important because testing in the design stage, even before attempting proofs, can be a cost-effective means of finding design flaws [Kem 85, DK 94].

The computational model for ASTRAL is based on nondeterministic state machines and assumes maximal parallelism, noninterruptable and nonoverlapping transitions in a single process instance, and implicit one-to-many (multicast) message passing communication.

Maximal parallelism assumes that each logical task is associated with its own physical processor, and that other

physical resources used by logical tasks (e.g., memory and bus bandwidth) are unlimited. In addition, a processor is never idle when some transition is able to execute. That is, a transition is executed as soon as its precondition is satisfied (assuming no other transition is executing). In addition, exported transitions require that a call be issued from the environment. If a new call is issued before the previous is handled, the most recent call supersedes the previous; i.e., ASTRAL does not provide any automatic ways of buffering external calls. The maximal parallelism approach was chosen on the basis of separating independent concerns; that is, first demonstrate that a design is satisfactory, then, and only then, consider the scheduling problem imposed by a particular implementation's limited resources. This approach, advocated in [FP 88] for real-time systems and in [CM 88] for parallel systems, allows for much cleaner designs than an approach that starts with scheduling concerns. A design based on the structure of the system rather than on its scheduling problems will usually be easier to maintain and/or modify. In addition, architectures that meet the maximal parallelism assumptions are becoming more prevalent.

Process cooperation, which involves both communication and synchronization, may be achieved in essentially two ways: either by data sharing or by message passing [BST 89]. The interface specification of RT-ASLAN is an example of modeling communication with shared data in a real-time specification language. At the implementation level, data sharing has obvious performance advantages and is, therefore, often used in current real-time systems. However, there is no apparent advantage in using data sharing at the specification level for describing process interactions at an abstract level. There are instead obvious disadvantages. For example, contention for shared data must be addressed in the specification, which implies that mutual exclusion also must be addressed. Furthermore, future real-time systems are likely to be less tightly-coupled than existing systems. For these reasons, in ASTRAL cooperation is modeled with implicit message passing rather than with data sharing. Implicit rather than explicit message passing was chosen to further simplify the design and to concentrate on the structure of the real-time system.

The specifics of the implicit multicast message communication model are that whenever a process instance starts executing an exported transition it broadcasts the start time and the values of the actual parameters to all interested processes (i.e., any process that may refer to the start time). When the transition is completed the end time as well as the new value of any exported variables that were modified by the transition are broadcast. Of course, any exported variables that are modified by a nonexported transition are also broadcast by the process when the transition completes execution. Thus, if a process is inquiring about the value of an exported variable while a transition is being executed by the process being queried, the value obtained is the value the variable had when the transition commenced. That is, the ASTRAL computation model views the values of all variables being modified by a transition as being changed by the transition in a single atomic action that occurs when the transition completes execution. These broadcasts are also assumed to be instantaneous.

## 3. ASTRAL LANGUAGE

In ASTRAL, a real-time system is described as a collection of state machine specifications, each of them representing a process type of which there may be multiple statically generated instances. There is also a global specification, which contains declarations for types and constants that are shared among more than one process type, as well as assumptions about the global environment and critical requirements for the whole system. Figure 1 presents the syntactic structure for an ASTRAL specification.

An ASTRAL *process specification* consists of a sequence of *levels*. Each level is an abstract data type view of the system being specified. The first (“top level”) view is a very abstract model of what constitutes the process (types, constants, variables), what the process does (state transitions), and the critical requirements the process must meet (invariants and schedules). Lower levels are increasingly more detailed with the lowest level corresponding closely to high level code.

The process being specified is thought of as being in various *states*, with one state differentiated from another by the values of the *state variables*, which can be changed only by means of *state transitions*. Transitions are described in terms of pre- and post- conditions by using an extension of first-order predicate calculus. Transition *entry conditions* describe the constraints that state variables must satisfy in order for the transition to fire, while *exit conditions* describe the constraints that are fulfilled by state variables after the transition has fired. An explicit nonnull duration is associated with each transition. Transitions are executed as soon as they are enabled (i.e., when their pre-condition is satisfied) assuming no other transition for that process instance is executing.

Every process can export both state variables and transitions; as a consequence, the former are readable by other processes while the latter are executable from the external environment. Inter-process communication is accomplished by broadcasting the values of exported variables and the start and end times of exported transitions.

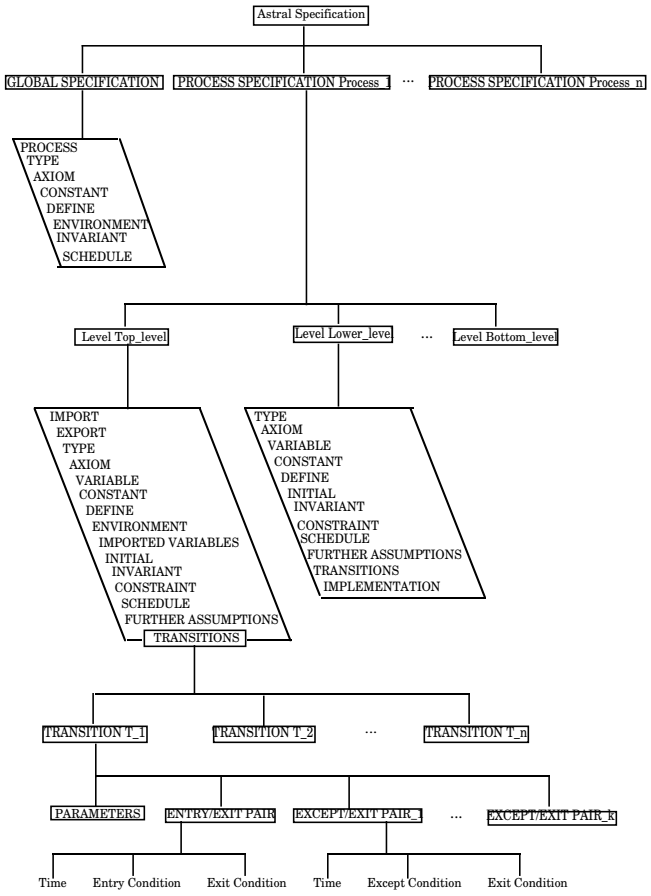


Figure 1: The ASTRAL hierarchy

In addition to specifying system state (through process variables and constants) and system evolution (through transitions), an ASTRAL specification also defines system critical requirements and assumptions on the behavior of the environment that interacts with the system. The behavior of the environment is expressed by means of *environment clauses*, which describe assumptions about the pattern of invocation of external transitions. Critical requirements are expressed by means of invariants and schedules. *Invariants* represent requirements that must hold in every state that may be reached from the initial state, no matter what the behavior of the external environment is, while *schedules* represent additional properties that must be satisfied provided that the external environment behaves as assumed.

In order to assure that an ASTRAL specification satisfies its requirements, it is necessary to generate and prove the appropriate proof obligations. ASTRAL proofs are divided into two categories: *intra-level* proofs and *inter-level* proofs. The former deal with proving that the specification of level i is consistent and satisfies the stated critical requirements for each of the processes, as well as for the global system. The latter deal with proving for each process type that the specification of level i+1 is consistent with the specification of level i. Details of the formal proof system can be found in [CKM 94, CKM 95].

To facilitate reuse and to simplify the construction of large and complex real-time systems ASTRAL provides the developer with a composition capability. The ASTRAL *compose clause* provides the necessary information to combine two or more ASTRAL system specifications (i.e., a global specification and its associated collection of process specifications) into a single specification of the combined system. The details of the composition clause, the process of composing ASTRAL specifications, and the necessary incremental proof obligations that need to be generated when composing two system specifications are presented in [CK 93].

## 4. THE ASTRAL SOFTWARE DEVELOPMENT ENVIRONMENT

In order to get designers to use formal methods to develop real-time systems it is necessary to provide them with an integrated set of tools for writing and analyzing their specifications. The ASTRAL Software Development Environment is intended to be an integrated set of design and analysis tools based on the ASTRAL formal framework. The tools that make up the support environment are a syntax-directed editor, a specification processor, a mechanical theorem prover, a specification testing component, and a browser kit.

The approach that was taken in building the ASTRAL software development environment was to first build an executive that integrates all of the component tools. This executive provides the user with a graphical user interface running under Unix with X-windows System 11 Release 5 with Motif. It includes extensive context-sensitive help facilities and the necessary optional features to get the level of guidance needed by a novice user while not hindering the experienced specifier. The system provides navigation windows, which allow the user to navigate through the specification in a hierarchical manner. A key design criteria for the environment was that the user should never need to switch between tools nor should there be any need for data exchange via temporary files. That is, the system is intended to be a fully integrated environment in which the user could change from specification writing to type checking to generating proof obligations with a mere button push. In its current state the software development environment contains stubs for the tools that are not yet ready to be integrated into the environment. Each of the component tools and their current status are discussed in the following subsections.

### 4.1 Syntax-Directed Editor

The ASTRAL syntax-directed editor aids the designer in constructing ASTRAL formal specifications. With this editor a user can build the specification in any order desired. When the user enters a specification component

the syntax is automatically checked. Figure 2 shows the user interface to the SDE. The core of the SDE is the navigation window located in the upper left portion. The navigation window displays the current specification and allows the user to hierarchically traverse it. By double clicking on a line of the displayed specification, a user can move "up" or "down" in the specification hierarchy. The specification displayed in the navigation window is the phone system specification of [CGK 96].

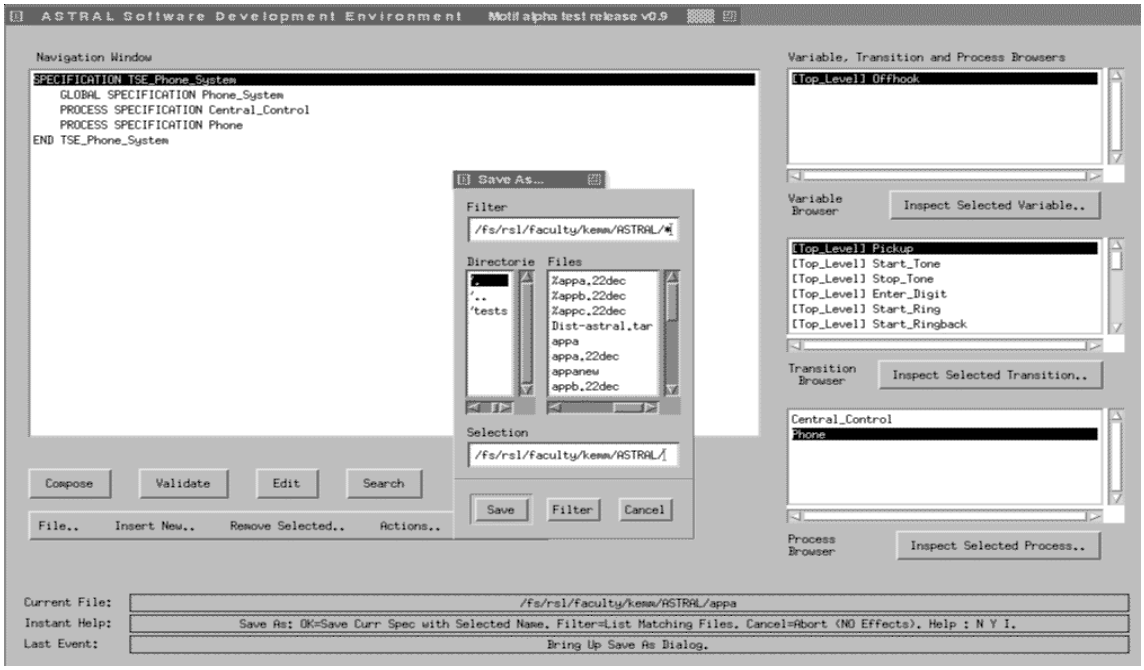


Figure 2: Screen Dump of Initial Navigation Window

At this level the user can see that there are two different process type specifications, in addition to the global specification. It also shows a popup window for saving the current specification and the help line explains the options. The screen dump in Figure 3 is the result of hierarchically navigating through the phone process and top level navigation windows and finally creating the Enter_Digit transition by selecting "transition" off the "Insert New" menu and then editing the newly created transition. The edit popup window has been selected to edit the exit assertion of the transition.

### 4.2 Specification Processor

The specification processor consists of a validation component and a verification condition generator (i.e., proof obligation generator). Because the user is allowed to enter the specification in any order, it is not always reasonable to perform all type checking immediately. Therefore, the user controls when the type checking is to be done. He/she can choose to type check the entire specification at any time by selecting the "Validate" button, which can be seen to the left of the bottom of the popup window in Figure 2.

The proof obligation generator component of the specification processor generates the necessary proof obligations to assure that each process specification is consistent with its invariants, constraints, and schedules, and that the properties of the collection of state machine specifications for the processes of the system is sufficient to assure the global invariant for the system and in some cases also the global scheduling constraints. These are known as the *intra-level* proof obligations. This tool also generates the necessary proof obligations to assure that a less abstract lower level specification for a process is a correct implementation of the parent specification that it implements. These are the *inter-level* proof obligations. The proof theories for both the intra-level and the inter-level proofs are defined in [CKM 94, CKM 95]. The proof obligations are generated by clicking on the "Actions" button, which can also be seen to the left of the bottom of the popup window in Figure 2, and choosing either intra-level or inter-level proof obligations. Because the theorem prover component has not yet been integrated in the environment, the resulting proof obligations are currently written to a file.

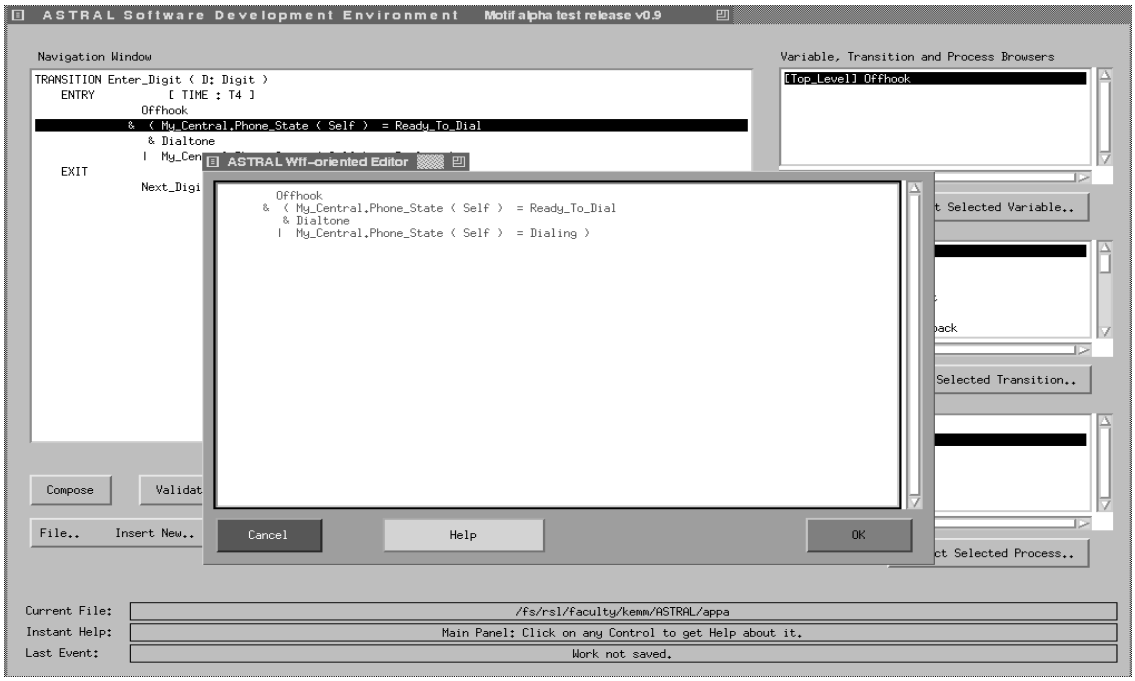


Figure 3: Screen Dump of Enter_Digit Navigation Window

### 4.3 Theorem Prover

A mechanical theorem prover component to discharge the proof obligations generated by the specification processor will also be integrated into the software development environment. However, it has been the experience of the Reliable Software Group that it is more efficient to build or adapt an existing theorem prover after one knows the types of theorems that need to be proved. For instance, when ASLAN was used in the MCC's Spectra software development environment [GBG 90, Ger 93] both the HOL [Gor 87] and Boyer and Moore [BM 90] theorem provers were adapted to prove the proof obligations that were generated by the ASLAN specification processor. More recently, the ASLAN specification processor was coupled to the Otter theorem prover [WM 92], which was developed at Argonne National Laboratory. Based on this experience, the integration of a theorem prover was delayed until a reasonable number of ASTRAL specifications had been written and the corresponding proof obligations had been generated. The proof obligations are currently being analyzed to determine what existing theorem prover(s) would be best to use.

### 4.4 Specification Testing Tool

In order to reduce the cost of developing reliable systems that also provide the desired functionality, it is necessary to provide a means to test the formal specifications early in the software life cycle. As was mentioned in the introduction, a scheme for translating ASTRAL formal specifications to TRIO was defined for an earlier version of the specification language. To test the feasibility of this approach a simple example specification that modeled a traffic light and the traffic on two intersecting streets was used. The complete specification was manually translated and the resulting specification was tested using the TRIO history checker, which was developed at the Politecnico di Milano [FM 94]. When testing the specification several problems with the ASTRAL specification, which revealed the potential for a traffic accident with the system as specified, were discovered. This experience lead to the decision to include an ASTRAL to TRIO translator and the TRIO-based history checkers as components of the software development environment.

However, now that a formal semantics for ASTRAL has been defined, the decision has been made to build on the Reliable Software Group's experience with directly executing formal specification languages symbolically. This experience includes implementing a symbolic execution tool for testing ASLAN specifications [DK 94], which is called Aslantest. This tool provides designers and users with a flexible means of animating system specifications to test for various functional requirements. The Reliable Software group has also designed and implemented a symbolic executor, called CASEX, for a verifiable subset of Ada, which includes tasking [HK 88]. This tool uses the interleaving approach, where the execution of component tasks are interleaved to model their concurrent execution. These tools are serving as a basis for the ASTRAL specification testing component, which is currently being designed.

### 4.5 Browsers

Finally, the software development environment contains three databases for analyzing inter-process and inter-transition relationships. These databases store information about variables, transitions, and processes of the current specification, and they are automatically updated and kept consistent every time the user edits anything in the ASTRAL specification. There are browsers associated with each of the databases and each of them has a dedicated window and a query button, which can be seen to the right of the navigation window in Figure 2. The browsers help the user to quickly determine what variables, transitions, or processes are affected by a change to some existing variable or transition. It is expected that the browsers will be very useful during the maintenance phase when the specifications are being modified.

## 5. RELATED WORK

MT [CHL 93] is an integrated development environment for the Modechart language, which incorporates components similar to those of the SDE. It includes the ability to hierarchically traverse specifications and invoke the editor on the displayed portion. MT's Consistency and Completeness Checker is similar in function to the SDE's validation procedure, performing a variety of static checks on the current specification. MT also provides a simulator which allows users to set up various conditions and display the results of a single execution path. Finally, MT has a verifier component to verify certain types of properties such as starvation and reachability.

STATEMATE [HLN 90] is another graphical environment for specifying reactive systems based on statecharts. It includes a number of different editors supporting statecharts, activity-charts, and module-charts which, like the SDE editor, check for syntactic errors immediately upon input. STATEMATE can also perform various consistency, completeness, and static logic tests at any time during a session. It also has a simulator which can be run either interactively or according to a program specified in a simulation control language. It can also generate a rapid prototype of the specified system in C or Ada code so that the simulator overhead can be avoided.

The Graphical Interval Logic Toolset [KRM 93] provides support for editing and verifying formulas written in Graphical Interval Logic. Like the SDE, the editor in the toolset is syntax-directed to prevent syntactic errors. Formulas can be easily composed to create more complex formulas. The toolset also incorporates a theorem prover which can, given a set of predicates from the user that supposedly imply a formula, search for and display an appropriate counterexample, if one exists.

## 6. CONCLUSIONS

ASTRAL has been used by both its developers and others to specify a number of interesting real-time systems. In [CKM 94] the results of using it to formally specify a CCITT system that consists of a packet assembler process and several input processes are reported. In [CGK 96] a phone system is composed with a switch to generate a wide-area phone system. The use of ASTRAL as a hardware description language was demonstrated in [BCF 91]. In that paper it was used to formally specify a checksum generator and a universal asynchronous receiver transmitter (UART) between a modem and a microprocessor. At Delft University of Technology (The Netherlands) ASTRAL was used to specify a robot control system [BBK 95]. These case studies demonstrate the expressiveness and the power of the language. They also show ASTRAL's usefulness for specifying varying types of real-time systems from basic hardware to complete communication systems. The packet assembler, the wide-area phone system, the standard railroad crossing and elevator examples, and an Olympic boxing scoring system have all been specified and validated using the SDE.

The SDE offers features that reduce errors and facilitate use throughout all stages of the specification development process. In the initial specification phase, the editor prevents syntax errors and the formatter enhances readability. In the middle phase, the validation function reports type errors, scoping errors, missing parameters, etc., and the VCG component generates the proof obligations needed to prove the specification correct with respect to its critical requirements. Finally, the browsers and compose/build features provide for easy maintenance and reuse of specifications.

## 7. ACKNOWLEDGMENTS

The authors would like to thank Richard Lee and Marco Mussini who worked on a prototype of the ASTRAL Software Development Environment.

## REFERENCES

[AK 85] Auernheimer, B. and R. A. Kemmerer, ASLAN User's Manual, TRCS84-10, Department of Computer Science, University of California, Santa Barbara, March 1985.

[AK 86] Auernheimer, B. and R.A. Kemmerer, "RT-ASLAN: A Specification Language for Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, September 1986.

[BST 89] Bal, H.E., J.G. Steiner, and A.S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, Vol. 21, No. 3, September 1989.

[BM 90] Boyer, R. and J. Moore, "A Theorem Prover for Computational Logic", *Proceedings of the 10th International Conference of Automated Deduction*, Kaiserslautern, Germany, July 1990.

[BBK 95] Brink, K., L. Bun, J. van Katwijk, and W.J. Toetenel, “Hybrid Specification of Control Systems”, F*irst IEEE International Conference on Engineering of Complex Computer Systems*, Ft. Lauderdale, Florida, November 1995.

[BCF 91] Buonanno G., A. Coen-Porisini, W. Fornaciari, “Hardware Specification Using the Assertion Language ASTRAL”, *Proceedings of the Advanced Research Workshop on Correct Hardware Design Methodologies*, Torino, Italy, June 1991.

[CM 88] Chandi, K.M. and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

CHL 93] Clements, P.C., C.L. Heitmeyer, B.G. Labaw and A.T. Rose, “MT: A Toolset for Specifying and Analyzing Real-Time Systems”, *Proceedings of the Real-Time Systems Symposium*, December 1993.

[CGK 96] Coen-Porisini, A., C. Ghezzi, and R.A. Kemmerer, “Specification of Realtime Systems Using ASTRAL”, Report no. TRCS 96-30, Department of Computer Science, University of California, Santa Barbara, California, July 1996.

[CK 93] Coen-Porisini, A. and R.A. Kemmerer, “The Composability of ASTRAL Realtime Specifications”, *Proceedings of the International Symposium on Software Testing and Analysis*, Cambridge, Massachusetts, July 1993.

[CKM94]. Coen-Porisini, A., R.A. Kemmerer and D. Mandrioli, “A Formal Framework for ASTRAL Intra-level Proof Obligations”, *IEEE Transactions on Software Engineering*, Vol. SE-20, No. 8, August 1994.

[CKM 95]. Coen-Porisini, A., R.A. Kemmerer and D. Mandrioli, “A Formal Framework for ASTRAL Inter-level Proof Obligations”, *Proceedings of the Fifth European Software Engineering Conference*, Barcelona, Spain, September 1995.

[CSK 94] Coen-Porisini, A., P. San Pietro and R. Kemmerer, "Formal Semantics Definition for ASTRAL", Report no. TRCS 94-15, Department of Computer Science, University of California, Santa Barbara, California, September 1994.

[DK 94] Douglas, J., and R.A. Kemmerer, "Aslantest: A Symbolic Execution Tool for Testing Aslan Formal Specifications," *Proceedings of the International Symposium on Software Testing and Analysis*, Seattle, Washington, August 1994.

[FP 88] Faulk, S.R. and D.L. Parnas, "On Synchronization in Hard Real-Time Systems," *Communications of the ACM*, Vol. 31, No. 3, March 1988.

[FM 94] Felder, M. and A. Morzenti, "Validating Real-Time Systems by History Checking TRIO Specifications", *ACM Transactions on Software Engineering and Methodologies*, Vol. 3 No. 4 , October 1994.

[Ger 93] Gerhart, S.L., "The MCC Formal Methods Transition Study: Technology Transfer for Complex Information Technology and Processes", *Proceedings of the IFIP TC8 Working Conference on Diffusion, Transfer and Implementation of Information Technology*, Pittsburgh, USA, October 1993.

[GMM 90] Ghezzi, C., D. Mandrioli, and A. Morzenti, "TRIO: A Logic Language for Executable Specifications of Real-Time Systems," *Journal of Systems and Software*, June 1990.

[Gor 87] Gordon, M., "*HOL: A Proof Generating System for higher order logic*", VLSI Specification, Verification, and Synthesis, Kluver, 1987.

[GBG 90] Greene, K., M. Bouler, S. Gerhart, and D. Russinoff, "SpecTra0.1 and SpecTra0.2: Demonstration and Research Issues", Tech. Rep. STP-EI-329-90, MCC Software Technology Program (Videotape), October 1990.

[HLN 90]. Harel, D. et al., “STATEMATE: A Working Environment for the Development of Complex Reactive Systems”, *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 4, April 1990.

[HK 88] Harrison, L.J. and R.A. Kemmerer, "An Interleaving Approach for the Symbolic Execution of Ada Tasking Programs," *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, Medford, New Hampshire, May 1988.

[Kem 85] Kemmerer, R.A., "Testing Software Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, January 1985.

[KRM 93]. Kutty, G., Y.S. Ramakrishna, L.E. Moser, L.K. Dillon and P.M. Melliar-Smith, “A Graphical Interval Logic Toolset foro Verifying Concurrent Systems”, *Proceedings of the Conference on Computer-Aided Verification*, June/July 1993.

[MS 94] Morzenti, A., D., and P. San Pietro, "Object Oriented Logic Specification of Time-Critical Systems," *ACM TOSEM*, Vol. 3, No. 1, pp. 56-98, January 1984.

[MMG 92] Morzenti, A., D. Mandrioli, and C. Ghezzi, "A Model Parametric Real-Time Logic," *ACM Transactions on Programming Languages and Systems*, Vol. 14, No. 3, July 1992.

[WM 92] Wos, L., W. McCune, "The application of automated reasoning to questions in mathematics and logic", *Annals of Mathematics and Artificial Intelligence*, Vol. 5, No. 2, pp. 321-370, May 1992

[Wir 77] Wirth, N., "Toward a Discipline of Real-Time Programming," *Communications of the ACM*, Vol. 20, No. 8, August 1977.

[Zwi 89] Zwiers, J., Compositionality, Concurrency, and Partial Correctness, *LNCS 321*, Springer Verlag, Berlin, 1989.