DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

The Concurrency Factory – Practical Tools for Specification, Simulation, Verification, and Implementation of Concurrent Systems

R. CLEAVELAND, J. N. GADA, P. M. LEWIS, S. A. SMOLKA,

O. SOKOLSKY, AND S. ZHANG

ABSTRACT. The *Concurrency Factory* is an integrated toolset for specification, simulation, verification, and implementation of concurrent systems such as communication protocols and process control systems. Two themes central to the project are the following: the use of *process algebra*, e.g., CCS, ACP, CSP, as the underlying formal model of computation, and the provision of *practical* support for process algebra. By "practical" we mean that the Factory should be usable by protocol engineers and software developers who are not necessarily familiar with formal verification, and it should be usable on problems of real-life scale, such as those found in the telecommunications industry.

The main features of the Concurrency Factory are *graphical* (VTView) and *textual* (VPL) *user interfaces*; a suite of *analysis routines* for automatic verification including a bisimulation and model checker; a *graphical simulator* for VTView specifications; and a *graphical compiler* that transforms VTView and VPL specifications into executable code.

1. Introduction

The *Concurrency Factory* is an integrated toolset for specification, simulation, verification, and implementation of concurrent systems such as communication

The Research of Rance Cleaveland was supported in part by NSF/DARPA grant CCR-9014775, NSF grant CCR-9120995, ONR Young Investigator Award N00014-92-J-1582, and NSF Young Investigator Award CCR-9257963. The research of Phil Lewis was supported in part by NSF grant CCR-9120995. The research of Scott Smolka was supported in part by NSF Grants CCR-9120995 and CCR-9208585, and AFOSR Grant F49620-93-1-0250DEF.

©0000 American Mathematical Society
0000-0000/00 $1.00 + $.25 per page

protocols and process control systems. The project, which is a joint effort between SUNY Stony Brook and North Carolina State University, officially started in Spring '92, and is currently supported by grants from NSF, ONR, and AFOSR. This paper can be viewed as a status report.

Two themes central to the project are the following: the use of *process algebra* [**Mil89, BK84, Hoa85**] as the underlying formal model of computation, and the provision of *practical* support for process algebra. By "practical" we mean that the Factory should be usable by protocol engineers and software developers who are not necessarily familiar with formal verification, and it should be usable on problems of real-life scale, such as those found in the telecommunications industry.

The main features of the Concurrency Factory are the following:

- A *graphical user interface*, VTView/VTSim [**Tre92, Jai93**], that allows the non-expert to design and simulate concurrent systems using process algebra. VTView is a graphical editor for hierarchically structured networks of finite-state processes, and VTSim is a sophisticated environment for the simulation and testing of VTView-constructed specifications. We are currently extending the GUI to allow processes to be embedded in states of other processes, thereby permitting compact specifications such as those found in statecharts [**Har87**].
- A *textual user interface* for the language VPL [**SS94b**], a simple language for concurrent processes that communicate values from a finite data domain. A VPL compiler translates VPL programs into networks of finite-state processes.
- A *suite of design and analysis routines* that currently includes a linear-time model checker for the alternation-free modal mu-calculus [**CS93**] and a bisimulation checker based on the algorithm of [**KS90**]. Care is being taken to ensure that these algorithms are efficient enough to be used on real-life systems. For example, we are investigating how these algorithms can be parallelized [**ZS92, ZSS94**], and made to perform incrementally [**SS94a**].
- A *graphical compiler* that transforms VTView and VPL specifications into executable code. Our current version produces Facile [**GMP89**] code, a concurrent language that symmetrically integrates many of the features of Standard ML [**Mil84**] and CCS [**Mil89**]. We are also considering adding a concurrent extension of C++ as another target language. The compiler relieves the user of the burden of manually recoding their designs in the target language of their final system.

The Concurrency Factory is written in C++ and executes under X-Windows, using Motif as the graphics engine, so that it is efficient, easily extendible, and

highly portable. It is currently running on SUN SPARCstations under SunOS Release 4.1. The basic organization of the system is depicted in Figure 1.

In terms of related work, the Concurrency Factory can be viewed as a next-generation *Concurrency Workbench* [**CPS93**], a toolkit for the analysis of finite-state concurrent systems specified as CCS expressions. The original version of the Workbench was implemented as a joint project between the University of Edinburgh, Sussex University, and the Swedish Institute of Computer Science (SICS). Enhanced versions are now being built at the University of Edinburgh, North Carolina State University, and SICS. All versions are written in ML and are primarily environments for experimenting with algorithms for analyzing concurrent systems, such as those for bisimulation, preorder, and model checking. Consequently the algorithm implementations on these versions are, in general, inefficient (see, for example, [**EF90**]), and the user interfaces are primitive. Moreover, neither provides implementation support. It is exactly these kinds of issues of practicality we seek to address with the Concurrency Factory.

Other process-algebra-based verification tools have also been developed. Auto/AutoGraph [**RdS90**] and Aldebaran [**Fer88**] support equivalence-based verification of concurrent systems, while PSF [**MV90**] and PAM [**Lin91**] provide users with an equational reasoning engine for establishing the correctness of systems. Winston [**MSGS90**] supports equivalence checking and includes a graphical editor for creating systems. TAV [**LGZ89**] includes facilities for stepwise-refinement of systems.

Still other tools, of which EMC[**CES86**] is the best-known example, enable users to formulate system specifications using temporal logic and check whether their systems meet such specifications. COSPAN [**HK90**] supports automata-based specifications, which users may compare for language inclusion, while tools like SPIN [**Hol91**] provide reachability analysis as their basic form of verification. Although not a verification tool, Statemate [**HLN+90**] arguably represents the most commercially successful formally based specification, simulation and implementation tool.

In the rest of this paper, we discuss the following aspects of the Concurrency Factory: its object-oriented design (Section 2), the graphical editor and simulator (Section 3), the analysis tools (Section 4), and the graphical compiler (Section 5). We conclude and present a number of directions of ongoing and future work in Section 6.

2. Object-Oriented Design of the Concurrency Factory

The Concurrency Factory is implemented in g++, the GNU project dialect of C++. We chose C++ for the following reasons: good compilers exist for virtually

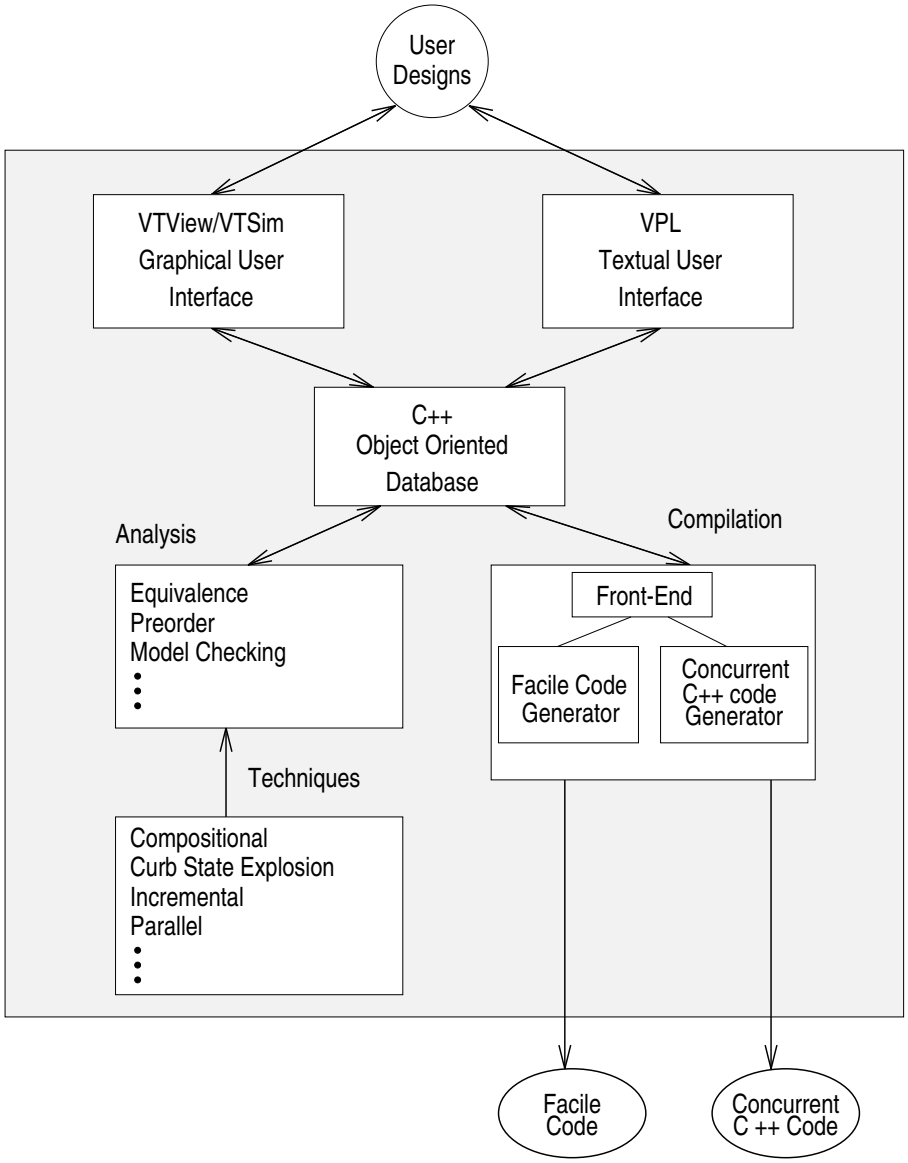

FIGURE 1. Basic organization of the Concurrency Factory.

every hardware platform, making the code highly portable; and the support the object-oriented approach provides for data abstraction and extendibility greatly facilitates the distributed development of software.

The class hierarchies for two of the main types of objects in the Factory, *processes* and *systems*, are depicted in Figure 2. Dashed arrows represent subclass relationships—e.g., `GProcess` is a subclass of `Process`—and solid arrows represents containment relationships—e.g., a `Process` object contains one or more `State` objects. To enhance readability, we have omitted some non-essential objects and containment arrows.

To understand the main ideas behind our object-oriented design consider first the class hierarchy for processes. A `Process` consists of one or more `States`, each of which has some number of incoming and outgoing `Transitions`. A `Process` is a *logical object* (the unshaded rectangles in Figure 2) in that it contains no information about how it should be displayed graphically. On the other hand, a `GProcess` is a `Process` endowed with additional such information and is an example of a *graphical object* (the shaded rectangles in Figure 2). A `GProcessEditor` contains a `GProcess` which it displays and can make changes to: it comes complete with a menu-bar of graphical editing functions suitable for a `GProcess`, e.g., create/delete a `GState` or `GTransition`.

Now consider systems. A `System` is either a `Network` or a `Process`. A `Network`, in turn, consists of one or more interconnected sub-`Systems`, thereby imparting a hierarchical structure to `Networks`. Sub-`Systems` are interconnected by embedding each of them in an `Interface` which defines the set of `Ports` a `System` can use to communicate with other `Systems`. Two `Systems` can communicate if they each possess a `Port` connected by a `Link` to a common `Bus`. An example system is depicted in Figure 3 of Section 3, which should help to clarify the preceding discussion.

One implementation note: we have made extensive use of the g++ generic *set* classes, in particular, the efficient AVL-tree implementation of sets. These are used, for example, to keep track of the `States` belonging to a `Process`.

We have paid particular attention in our design to the encapsulation of processes and networks as abstract data types. As an illustration of this, consider again the class `Process`. The prototypes for two of the methods we provide for this class are as follows:

```
Transition* Process::AddTransition(String &act,
            State *src, State *tgt)

void Process::DeleteTransition(Transition *t)
```

The first method creates a new `Transition` object from a source state `src` to

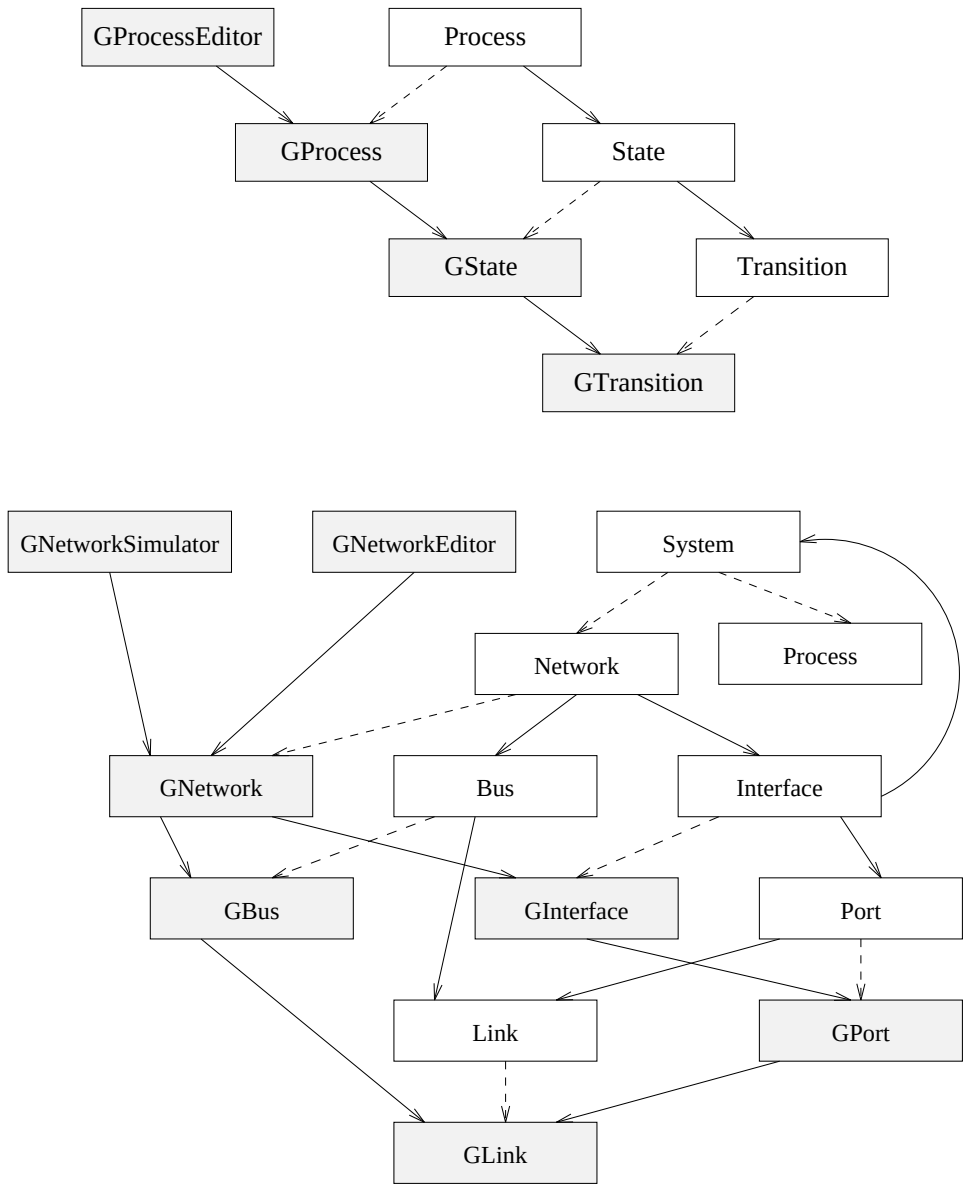

FIGURE 2. Class hierarchies for the process and system objects.

a target state `tgt` labeled by string `act`. The newly created transition will automatically be added to the source state's set of outgoing transitions and to the target state's set of incoming transitions. Note that this is the only way a user can add transitions to these sets. The net result is a well-defined `Process` object with the new transition added, as desired. Similarly, the second method deletes a `Transition` object and all related objects so that the resulting `Process` is again in a well-defined state.

3. Graphical Editor and Simulator

The graphical user interface of the Concurrency Factory consists of two main components, VTView and VTSim (VT stands for *Verifier's Toolkit*). VTView [**Tre92**] is a graphical tool supporting the design of hierarchically structured systems of communicating tasks expressed in GCCS, a graphical specification language. GCCS provides system builders with intuitive constructs (buses, ports, links, a subsystem facility, etc.) for designing systems, and it supports both top-down and bottom-up development methodologies. As user designs are entered using VTView, an internal representation is created by invoking the appropriate methods associated with process and network objects.

In contrast with other graphical languages [**Har87, Mar89**], GCCS is designed to model systems in which processes execute asynchronously (although communication between processes is synchronous). The language is equipped with a formal semantics in the form of a structural operational semantics, à la Plotkin [**Plo81**] and [**Mil89**]. The semantics has been "implemented" in the Factory in the form of methods that determine the set of transitions that are possible for a network or a process from a given state. Both the graphical simulator and the method that produces the global automaton from a network of process rely fundamentally on these methods. By encapsulating the semantics of VTView objects, all tools within the Factory are guaranteed to interpret processes and networks consistently.

VTSim [**Jai93**] permits users to simulate graphically the execution of GCCS systems built using VTView. The tool provides both interactive and automatic modes of operation, and it also includes features such as breakpoints and reverse execution. It also enables users to view the simulated execution of a system at different levels in the structure; one can either choose to observe the simulation at the interprocess level and watch the flow of messages, or one can look at individual processes in order to see why messages are sent when they are.

To illustrate the look-and-feel of VTView and VTSim, consider the well-known Alternating Bit Protocol (ABP). ABP consists of a sender and receiver process communicating over an unreliable medium, also modeled as a process. Figure 3 depicts the network structure of the protocol, and the transition system for the

sender process. In the network structure diagram, processes are represented by small labeled rectangles, interfaces are the larger rectangles that enclose the process rectangles, ports are the circles appearing along the perimeter of interfaces, and ports are connected to buses via links. Note that the depth of the hierarchy for ABP is two as all interfaces enclose processes; i.e., the ABP network does not contain any sub-networks.

4. Verification Tools

This section describes the verification tools currently provided by the Concurrency Factory. Other tools are planned, and these are discussed in Section 6. We begin with the *global automaton builder*, which given a network of processes, recursively constructs the reachable global state space of the network. Like VTSim, the global automaton builder invokes the network semantic methods described above to orchestrate the construction procedure. The main role played by the global automaton builder is that it allows the other verification tools to operate on either a process or a network; in the latter case, the global automaton builder would be invoked before verification begins to convert the network into what is essentially a process. In Section 6 we discuss *compositional* approaches to verification. These techniques attempt to avoid the often costly construction of a network's global state space.

The main verification tool provided by the Factory is a linear-time model checker for the alternation-free modal mu-calculus. The *modal mu-calculus* is a highly expressive logic that can be used to specify safety, fairness, and liveness properties of concurrent systems represented as labeled transition systems (LTSs). Syntactically, it consists of atomic propositions, standard logical connectives $\wedge$ and $\vee$, dual modal operators $\Box$ (necessarily) and $\Diamond$ (possibly), and dual fixpoint operators μ (least fixed point) and ν (greatest fixed point). The *alternation-free* fragment of the modal mu-calculus consists of those formulas for which the "level" of mutually recursive greatest and least fixed points is one.

Given an LTS, which in Factory parlance is a system (see Section 2), and a formula of the modal mu-calculus, the *model checking* problem is to determine whether the LTS is a model of the formula. The algorithm reported in [**CS93**] solves this problem, for the alternation-free fragment, in time linear in the size of the LTS and the formula, and is the basis for the Factory's model checking tool.

We have also implemented an *incremental* version of the model checker, which can be used to substantial advantage when a small, incremental change is made to the LTS or the formula after having invoked the original, "start-over" model checker. By reusing previously computed results, the incremental model checker often runs much faster than its start-over counterpart. In some cases, the incre-

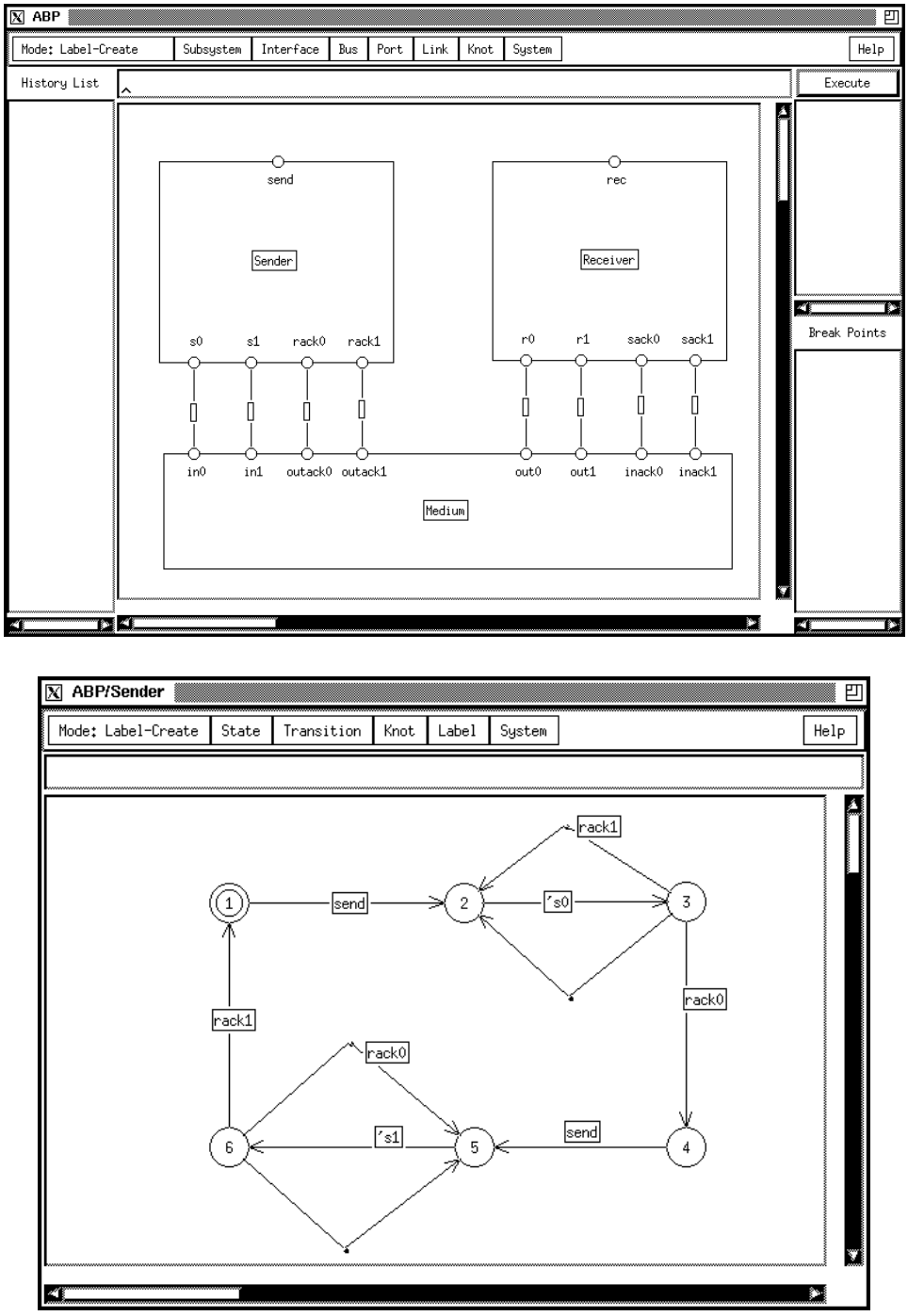

FIGURE 3. Network architecture of ABP and transition system of sender process.

mental model checker requires only time constant in the size of the system being analyzed, and never more than twice the time taken by the start-over algorithm.

Another verification tool provided by the Factory checks whether two systems are *bisimilar*, which essentially means they are indistinguishable to an external observer [**Mil89**]. The tool is based on the *generalized partitioning* algorithm of [**KS90**], which runs in time $O(mn)$, where m is the total number of transitions and n is the total number of states in the systems being compared.

In order to improve the performance of the bisimulation checker, we have implemented a parallel version that runs on an Intel iPSC/860 parallel computer (processors on this machine are interconnected as a hypercube). This has led to significant speedups, which are reported in [**ZS92**].

5. Graphical Compiler

The graphical compiler translates GCCS specifications of concurrent systems into executable code, thereby relieving the user of the burden of manually recoding their designs in the target language of the final system. The graphical compiler would typically be invoked after the user has utilized the simulation and verification facilities of the Factory to help ensure that the specifications are correct.

The current version of the graphical compiler produces Facile [**GMP89**] code, a language that symmetrically integrates many of the features of Standard ML [**MTH90**] and the process algebra CCS, on which GCCS is based. Because of its CCS-like operators, the translation into Facile is very natural. Moreover, the current release of Facile, Facile Antigua, runs on LANs and WANs on top of TCP/IP, making it a true distributed programming language.

GCCS specifications mainly address the synchronization aspects of concurrent systems. Data-oriented issues such as the contents of messages are not specifically treated. After generating Facile code from VTView specifications, the user can directly code the data-oriented parts of the system in the ML subset of Facile.

6. Conclusions and Future Work

We have described the Concurrency Factory, a graphical environment that supports the following system development tasks: specification (VTView), simulation (VTSim), verification (model and bisimulation checking), and implementation (Facile graphical compiler). Much work remains to be done on the project, including the following.

Applications. As a first step, we have used the Concurrency Factory graphical user interface GCCS to specify the Alternating Bit Protocol (see Figure 3). The specification has been submitted to the graphical compiler and the resulting Facile code has been executed on a network of workstations. Similar plans are underway for the IEEE 802.5 token ring protocol.

We are also in the process of using the Factory to debug the *'i' protocol*, a communication protocol used in the GNU UUCP V1.04 system-to-system file copy utility. Gene Stark, a colleague at SUNY Stony Brook, has discovered a livelock in the protocol, and we are now encoding the protocol in VPL. When finished, we will submit the protocol to the Factory's modal mu-calculus model checker in an attempt to unearth the source of livelock.

Another potential application involves a program supplied to Stony Brook by a Long Island-based company, Parker-Hannifin, a division of Gull Electronics. The Parker-Hannifin program manages the display of the Stabilizer Position Indicator (SPI) that is being installed in the cockpits of Boeing 777 aircraft. To handle the SPI and other intended applications, we are extending the Factory's user interface and analysis techniques to handle real-time specifications.

State Refinement. In [**US93, US94b, US94a**], we introduced a new process algebraic operator that allows one process to be embedded in the state of another process. State refinement captures the key ideas of hierarchy and concurrency indigenous to Statecharts [**Har87**]; the end result is that system specifications are often much more compact and have a natural graphical representation. We are currently adding state refinement to VTView/VTSim so that statechart-like behaviors may be specified and verified within the Concurrency Factory environment.

Parallel Algorithms. In Section 4 we mentioned that we have implemented a parallel version of our bisimulation checker on the Intel iPSC/860. Currently, this routine must be invoked via a remote procedure call. To obtain a more integrated resource, we are re-implementing the parallel bisimulation checker using Solaris threads.

We have also been investigating how to parallelize the model checker. Our results [**ZSS94**] show that the problem of model checking in the modal mu-calculus is P-hard — meaning that there is probably something inherently sequential about the problem — even for the alternation-free fragment of modal mu-calculus. However, we were able to design efficient NC parallel algorithms for certain restricted cases, which may provide useful parallel heuristics in the general case.

Compositional Algorithms. The essential idea in a *compositional* algorithm is to avoid computing the global state space of a network of processes during verification. Rather, component processes are analyzed largely in isolation and the results of these analyses are then combined to achieve an analysis of the given network. We are currently investigating how to modify our model checker so that it works compositionally (see, for example, [**ASW94**]).

Another related possibility is to *localize* model checking, as in [**Cle90, SW91**]. In this case, verification is carried out with a particular system state in mind and only as much of the global state space as is needed to solve the problem with respect to this state is explored.

User-Friendly Testing Language. The modal mu-calculus is the logic used in conjunction with the Factory's model checker to specify correctness conditions of concurrent systems. Although highly expressive, formulas of the logic are often difficult for humans to understand even when the alternation depth is one. For this reason, we are developing a more user-friendly testing language, T, based on tree expressions. Intuitively a specification in T is built from subtrees that specify fragments of the overall system behavior. Formulas of this language will be automatically translated into the modal mu-calculus before being input to the model checker.

Extensions to GCCS. The current version of GCCS provides users with a notation for modeling processes that synchronize. In order to handle other forms of process interaction (e.g., asynchronous communication, broadcasting, and message-passing) we are investigating appropriate extensions to the core language. Our early work is focusing on adding different kinds of buses, which are the communication "media" supported by the tool, to support different communication disciplines. We are also looking into the use of the network level of GCCS as a *co-ordination language* in which users could embed programs written in different languages as processes, with the GCCS network layer co-ordinating the communication among processes.

Using GCCS as a co-ordination language will require us to extend the range of the Factory's analysis techniques from basic process algebra expressions to those with data. To this end, we are exploring how to integrate the well-known technique of *abstract interpretation* into our equivalence and model checking algorithms [**CR94**].

Textual Interfaces. Early feedback on the Factory indicates that some users for some applications would be happier with a textual rather than graphical user interface. To address this need, we have designed a simple language, called VPL, for concurrent processes that communicate values from a finite data domain. A compiler is written that will translate VPL programs into a sequence of method

invocations. Executing these calls will result in the creation of a network object representing the source program.

The unit of specification in VPL is a process, which represents a sequential machine. Processes communicate through bi-directional channels using the standard communication primitives $c!expr$, which outputs the value of expression $expr$ on channel c, and $c?v$, which inputs a value from channel c into variable v. VPL provides high-level control constructs for processes, including conditional execution, iteration and nondeterminism; and supports simple data types in the form of bounded-size integers and composite ones: arrays and records. Processes can be composed in parallel and communicate through typed channels.

Since GCCS does not yet support value passing, translation of VPL programs into GCCS networks must employ some kind of conversion from a value-passing calculus into a basic one. Integer variables are translated into separate processes that execute in parallel with the process representing the control part and on request communicate their value to the control part (which corresponds to reading a variable).

VPL is currently being used to describe the GNU UUCP i-protocol mentioned above. A similar textual interface is in the planning stages for LOTOS, an official ISO (International Standards Organization) formal description language for communications protocols.

Object-Oriented Database. We are in the process of equipping the Concurrency Factory with an object-oriented database system, which will allow objects to be accessed efficiently and uniformly in a persistent manner. It will also allow concurrent access to objects and provide sophisticated version control. The database system we are using is E, a persistent extension to GNU g++ developed at the University of Wisconsin-Madison as part of the Exodus project.

Acknowledgements

Gene Stark helped to design and implement VPL while Daniel Haenle implemented the compiler to Facile.

References

[ASW94] H. R. Andersen, C. Sterling, and G. Winskel. A compositional proof system for the modal μ-caluclus. In *LICS '94*, July 1994.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60:109–137, 1984.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Cle90] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–747, September 1990.

[CPS93] R. Cleaveland, J. Parrow, and B. U. Steffen. The concurrency workbench: A sematics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1), 1993.

[CR94] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *Proceedings of CONCUR '94 — Fifth International Conference on Concurrency Theory*, August 1994.

[CS93] R. Cleaveland and B. U. Steffen. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2, 1993.

[EF90] P. Ernberg and L. Fredlund. Identifying some bottlenecks of the concurrency workbench. Technical Report SICS/T-90/T9002, Swedish Institute of Computer Science, Kista, Sweden, April 1990.

[Fer88] J. C. Fernandez. *Aldebaran: Un Systeme de Verification par reduction de processus communicants.* PhD thesis, Universite de Grenoble, France, 1988.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmertric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2), 1989.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HK90] Z. Har'El and R.P. Kurshan. Software for analytical devlopment of communications protocols. *AT&T Technical Journal*, 69(1):45–59, January/February 1990.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Traktenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, London, 1985.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Jai93] S. Jain. VTSIM: A graphical simulator for finite-state networks. Master's thesis, Department of Computer Science, North Carolina State University, 1993.

[KS90] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 90.

[LGZ89] K. G. Larsen, J. C. Godskesen, and M. Zeeberg. Tav - tools for automatic verification — users manual. Technical Report R 89-19, Department of Mathematics and Computer Science, Aalborg University, 1989.

[Lin91] H. Lin. A process algebra manipulator. Technical Report 2/91, University of Sussex, February 1991.

[Mar89] F. Maraninchi. Argonaute, graphical description, semantics and verification of reactive systems by using a process algebra. In *Proc. CAV '89*, volume 407 of *Lecture Notes in Computer Science*, pages 38–53, Grenoble, June 1989. Springer-Verlag.

[Mil84] R. Milner. A proposal for standard ML. Technical Report CSR-157-83, Department of Computer Science, University of Edinburgh, 1984.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[MSGS90] J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro. Winston: A tool for hierarchical design and simulation of concurrent systems. In *Proceedings of the Workshop on Specification and Verification of Concurrent Systems.* Springer-Verlag, Workshops in Computing Series, 1990.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MV90] S. Mauw and G. J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[RdS90] V. Roy and R. de Simone. Auto/Autograph. In *DIMACS Workshop on Computer-*

Aided Verification, pages 235–250, Piscataway, New Jersey, July 1990. American Mathematical Society.

[SS94a] O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. To appear in CAV'94, June 1994.

[SS94b] O. Sokolsky and E. Stark. VPL – a value-passing language for specifying communication protocols. Technical report, Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY, 1994.

[SW91] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1), 1991.

[Tre92] V. Trehan. VTVIEW: A graphical editor for hierarchical networks of finite-state processes. Master's thesis, Department of Computer Science, North Carolina State University, December 1992.

[US93] A. C. Uselton and S. A. Smolka. State refinement in process algebra. In *Proc. Second North American Workshop on Process Algebra, NAPAW '93*, Ithaca, NY, August 1993.

[US94a] A. C. Uselton and S. A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *Proceedings of CONCUR '94 — Fifth International Conference on Concurrency Theory*, Uppsala, Sweden, August 1994.

[US94b] A. C. Uselton and S. A. Smolka. A process-algebraic semantics for statecharts via state refinement. In *Proceedings of PROCOMET '94*. North Holland/Elsevier, 1994.

[ZS92] S. Zhang and S. A. Smolka. Efficient parallelization of equivalence checking algorithms. In M. Diaz and R. Groz, editors, *Proceedings of FORTE '92 – Fifth International Conference on Formal Description Techniques*, pages 133–146, October 1992.

[ZSS94] S. Zhang, O. Sokolsky, and S. A. Smolka. On the parallel complexity of model checking in the modal mu-calculus. In *LICS '94*, July 1994.

R. Cleaveland, J. N. Gada
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
USA

E-mail address: {rance, jngada}@science.csc.ncsu.edu

P. M. Lewis, S. A. Smolka, O. Sokolsky, S. Zhang
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
USA

E-mail address: {pml, sas, oleg, szhang}@cs.sunysb.edu