GCCS: A Graphical Coordination Language for System Specification

Rance Cleaveland, Xiaoqun Du, Scott A. Smolka

Department of Computer Science, SUNY at Stony Brook
Stony Brook, NY 11794–4400, USA
{rance,vicdu,sas}@cs.sunysb.edu

Abstract. We present GCCS, a graphical coordination language for hierarchical concurrent systems. GCCS, which is implemented in the Concurrency Factory design environment, represents a coordination model based on process algebra. Its coordination laws, given as a structural operational semantics, allow one to infer atomic system transitions on the basis of transitions taken by system components. We illustrate the language's utility by exhibiting a GCCS-coordinated specification of the Rether real-time ethernet protocol. The specification contains both graphical and textual components.

1 Introduction

As defined in [Cia96], a *coordination language* or *model* provides a framework for describing how independent "agents" may interact. A coordination model should therefore dictate a number of *coordination laws* describing how agents use the given coordination media (semaphores, monitors, channels, tuple spaces, etc.) to coordinate their activities.

In this paper, we present a coordination model based on *process algebra* whose coordination laws are given as a *structural operational semantics* [Plo81]. The process algebra we utilize for this purpose is called *Graphical Calculus of Communicating Systems* (GCCS), a graphical version of Milner's value-passing CCS [Mil89] for specifying hierarchical networks of communicating processes. The coordination media in our case consist of *channels*, and the corresponding coordination primitives are bi-party synchronous message passing. GCCS's operational semantics is given by a small collection of *semantic rules* that allow one to infer atomic system transitions on the basis of transitions taken by system components.

The main virtue of using process algebra as a coordination language is that *any programming language or design notation possessing a "compatible" operational semantics can, in principle, be incorporated into the resulting coordination framework.* Further, the presence of a formal operational semantics ensures that any system implemented using the coordination language may be simulated in a precise mathematical sense and formally verified, using, for example, equivalence and model checking. Moreover, the graphical and hierarchical nature of

GCCS aids in the visual comprehension of the software architecture [S+95] of the system under construction.

As a demonstration of these points, we have implemented a GCCS-based coordination framework in the Concurrency Factory design environment for concurrent and distributed systems [CGL+94,CLSS96a,CLSS96b]. In that incarnation of the framework, subsystems may be specified in VPL, the Concurrency Factory's textual design notation, or in the subset of GCCS for graphically rendering communicating state machines. Both of these notations possess an underlying operational semantics based on the notion of transitions labeled by atomic actions, and are thus compatible with the operational semantics of GCCS.

The Concurrency Factory also provides facilities for automatically generating executable code from GCCS-based designs. To date we have implemented code generators for Java, Ada 95, and C. These code generators target a variety of execution platforms including those based on threads, RPC, and sockets. An important aspect of the Concurrency Factory's code generators is that code is produced for the entirety of the coordination layer of the submitted GCCS design, covering all aspects of interprocess synchronization and communication. At the process level, code is generated for constructs involving basic (sequential) flow of control and specified data structures. Place-holders are provided in the generated code for application-specific code not captured in the GCCS design.

The Concurrency Factory has been applied to a number of real-life case studies, including [DMN+97,DSC99,DSSW99,DDR+99]. The case study reported in [DMN+97,DSC99]—involving the formal specification and verification of the Rether real-time software-based ethernet protocol for multimedia applications—in particular, makes substantial use of GCCS's coordination features. We thus use it here to illustrate our approach.

Traditional approaches to coordination deploy a coordination language during *coding* stages of system development; i.e. coordination languages have typically been programming languages as well. Some popular coordination languages of this nature include Linda [CG89], JavaSpaces [Sun98], and TSpaces [WMLF98]. GCCS, on the other hand, is intended primarily as a *design* language. For several reasons, however, this difference is not as large as it may first seem. First, as mentioned above, any language with a GCCS-compatible operational semantics can be incorporated into the GCCS coordination framework; thus, in principle, this includes programming languages (or some suitable subsets thereof). Secondly, it is possible to automatically generate executable code from GCCS coordinated designs, and this is the approach taken in the Concurrency Factory. A benefit of a design-oriented coordination model is that coordinated system descriptions may first be submitted to formal verification and simulation before being "compiled" into executable code, thereby affording system designers greater confidence in the final product.

In other related work, Bergstra and Klint [BK98] propose ToolBus, a software coordination architecture that uses scripts and pattern matching for describing the interaction among software components. The semantics of the scripts has been given in terms of the process algebra ACP. In [AG97] and [BCD99], architectural description languages with semantics based on process algebras have been proposed. Process algebraic techniques such as refinement and weak bisimulation equivalence are used to analyze architectural compatibility and architectural conformity of software systems.

Process algebra has also been used by Busi, Gorrieri, Zavattaro and coworkers as a semantic basis for a number of recently proposed coordination languages [BGZ98,BGZ00,BZ00,AdBB+00], including Linda, JavaSpaces, TSpaces, and Manifold [AHS93]. This ground-breaking work allows one to clarify possible ambiguities in the definitions of these languages, discuss possible implementation choices, compare expressive power, and formally reason about programs written in these languages. In contrast, our approach uses process algebra and operational semantics *directly* as a coordination framework, rather than to define the semantics of independently proposed coordination languages. Moreover, any language equipped with a process-algebraic semantics becomes a candidate component-specification language for the GCCS framework, provided that the algebra's operational semantics are GCCS-compatible. Thus the coordination languages investigated by Gorrieri, Zavattaro, et al. are such candidates.

The rest of the paper is organized as follows. Section 2 gives an overview of the Concurrency Factory. Sections 3 and 4 describe the syntax and operational semantics of GCCS and VPL, respectively. Section 5 discusses some of the implementation issues surrounding GCCS as a coordination language. Section 6 presents the Rether example, while Section 7 gives our concluding remarks and future research directions.

2 The Concurrency Factory

The Concurrency Factory is an integrated toolset for specification, simulation, verification, and implementation of concurrent and distributed systems such as communication protocols and process control systems. The Factory uses process algebra as its underlying formal model of systems, and model checking and equivalence checking as its primary verification techniques. The goal of the project is to make process algebra and model checking both: (1) accessible to system engineers who might not be familiar with formal verification; and (2) applicable to the kinds of complex systems one is likely to encounter in real-life applications. Both considerations factored heavily into our design of the description languages. In support of (1), we wanted a graphical "architectural" language with constructs similar to the ones engineers use informally in the early design of a system. In support of (2), the notation needed to be modular and to support efficiently the manipulation of data.

These observations led to the inclusion of the following key features of the Concurrency Factory.

- A graphical editor and viewer supporting GCCS as a coordination language for hierarchically structured networks of communicating processes.

- A textual user interface for the VPL, a language based on Milner's CCS for describing systems of concurrent processes that communicate values. Like GCCS, VPL systems may be multi-layered or hierarchical.
- An interactive graphical simulator for GCCS that allows the user to witness GCCS-based interactions between coordinated subsystems, and to view these interactions at any level of the network hierarchy—simultaneously if so desired.
- A set of object interfaces for representing system specifications. Both GCCS specifications and compiled VPL programs have internal representations that, although quite different, share a high-level interface. Methods "at the interface" include a set of "semantic routines" that capture the operational semantics of these objects. The coordination mechanisms use these methods in order to calculate the interactions allowed between objects. The methods also provide a uniform interface to the Factory's verification and simulation routines.
- A suite of design and analysis routines that includes efficient model checkers for the modal mu-calculus, an expressive branching-time temporal logic, and a bisimulation checker.
- A graphical compiler that transforms GCCS and VPL specifications into executable Java or ADA 95 code.

The architecture of the Concurrency Factory is shown in Fig 1. The Factory is written in C++ and executes under X-Windows, using Tcl/Tk as the graphics engine, so that it is efficient, easily extendible, and highly portable.

3 Syntax and Semantics of GCCS

3.1 GCCS Syntax

GCCS contains two sublanguages: the *coordination layer* and the *process layer*. The coordination layer supports the definition of *systems*, where systems may be processes or *networks* of subsystems. Note that networks are hierarchical, as subsystems may themselves be networks, and that processes constitute the "leaves" of the hierarchy.

Within a network, subsystems may be topologically interconnected in the following manner. Each system possesses an *interface* defining the collection of *ports* the system can use can use to communicate with other systems. At the root-level of the hierarchy, a system's ports define its interface to the outside world. Ports are labeled and the communication actions that involve a given port are named in accordance with the port's label. A communication pathway between subsystems may then be formed by connecting ports of the systems to a common *bus*, the communication/coordination mechanism of GCCS. Connections between ports and buses are called *links*. Buses, like ports, may also be labeled, and communication actions occurring over ports linked to a labeled bus are renamed to the bus's label.

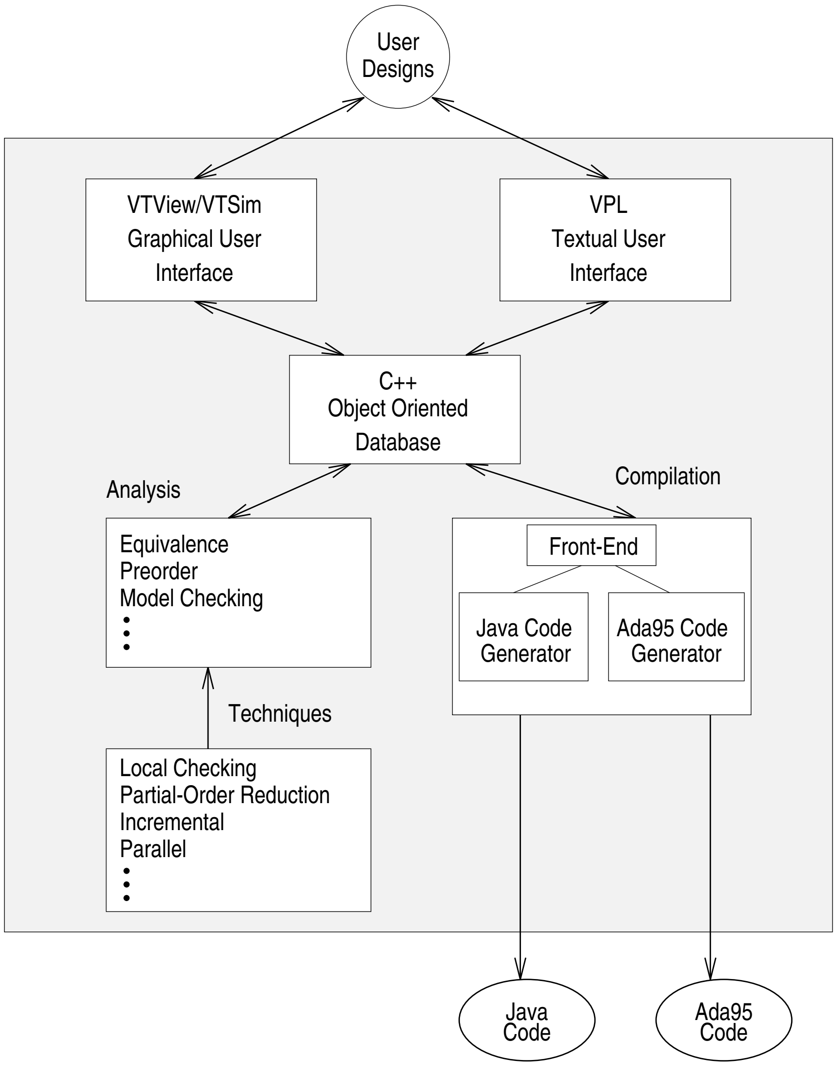

Fig. 1. The architecture of the Concurrency Factory.

Connections to buses may be multi-way, i.e. two or more systems may contain links to the same bus. However, communication over buses is bi-party and synchronous, meaning that any actual communication involves exactly two systems and requires a rendezvous. Besides its possible involvement in a synchronized communication, a communication action taken by a system over a particular port can be redirected to a port with a matching label one level higher in the system hierarchy.

The GCCS network appearing in the top middle panel of Figure 2 illustrates the above concepts. The example, corresponding to the aforementioned Rether protocol (discussed more fully in Section 6), involves a root system (Rether) composed of three subsystems: `adm_ctrl`, `band`, and `nodes`. `adm_ctrl` and `band` are processes (leaves) while `nodes` is a system whose coordination structure can be viewed by zooming in on its icon. As Figure 2 illustrates, interfaces are rendered graphically in GCCS as rectangles and ports are drawn as labeled blackened circles along rectangle perimeters. Line segments are used to link ports to buses.

The coordination-layer constructs of GCCS allow one to graphically depict the hierarchical coordination infrastructure of the specified system. The process layer of GCCS contains graphical constructs for specifying GCCS processes, the leaves of a GCCS system hierarchy. As described in Section 4, processes can also be specified textually using VPL. A GCCS process is essentially a communicating state machine or labeled transition system, consisting of states and state transitions. Each transition has a unique source state and target state, and is labeled by a communication action. As described above, communication actions are named in accordance with the label of the port over which they occur. A transition can also be labeled by τ, representing a CCS-like internal action [Mil89].

Also, as in CCS, communication actions are either input actions or output actions. Output actions are barred to distinguish them from input actions. Thus the action $\overline{a}$ represents an output action taken on behalf of a process over port a. Communication actually occurs in GCCS when an input action with a given name synchronizes with an output action of the same name. Thus, for instance, input action a can synchronize with output action $\overline{a}$. In "pure" GCCS, which for simplicity we focus on in this paper, communication is data-less, involving only the exchange of a signal (named a in the example). The version of GCCS currently implemented in the Concurrency Factory does support data exchange; the language has a type system consisting of bounded width integers together with record and array constructs, and ports and buses must have types associated with them. This support for data allows a seamless integration with VPL, which provides support for value passing between processes.

3.2 GCCS Semantics

The formal semantics of GCCS is based on labeled transitions systems. To define the semantics precisely we follow the approach used for other graphical languages such as Statecharts; we first introduce a term-language for GCCS and then define the semantics on these terms. We begin with a few preliminary definitions.

- $\mathcal{A}$ is an infinite set of *port names* not containing τ or ξ. $a, a', \ldots$ range over $\mathcal{A}$.
- $\mathcal{B} = \mathcal{A} \cup \{\xi\}$ is the set of *bus names*; ξ is the internal bus. $b, b', \ldots$ range over $\mathcal{B}$.
- $\mathit{InputActions} = \mathcal{A}$ is the set of *input actions*.
- $\mathit{OutputActions} = \{\overline{a} | a \in \mathcal{A}\}$ is the set of *output actions*. $\overline{a}, \overline{a'}, \ldots$ range over $\mathit{OutputActions}$, and $l, l', \ldots$ range over $\mathit{InputActions} \cup \mathit{OutputActions}$.
- $\mathit{Act} = \mathit{InputActions} \cup \mathit{OutputActions} \cup \{\tau\}$, where τ is an internal action, represents the set of actions. $\alpha, \beta, \ldots$ range over Act.
- $\mathit{port} : \mathit{InputActions} \cup \mathit{OutputActions} \rightarrow \mathcal{A}$ maps each non-τ action l to its associated port.

We now define the term language GCCS as follows.

- A *system* S is either a process or a network.
- A *process* is a tuple $\langle Q, \rightarrow, q_0, q\rangle$, where Q is a finite set of states, $\rightarrow \subseteq Q \times \mathit{Act} \times Q$ is a finite set of labeled transitions, $q_0 \in Q$ is the initial state, and $q \in Q$ is the current state. The set of processes is represented by Proc.
- A *network* is a tuple $\langle \overline{N}, B, L\rangle$ whose components have the following form.
  1. $\overline{N} = \langle\langle S_1, I_1\rangle, \ldots, \langle S_n, I_n\rangle\rangle$ is a finite sequence of *system/interface* pairs, where each $I_i \subseteq \mathcal{A}$ is required to be finite. We use $|\overline{N}| = n$ to denote the length of $\overline{N}$.
  2. $B \subseteq \mathcal{B}$ is a finite set of buses.
  3. $L \subseteq \{1, \ldots, |\overline{N}|\} \times \mathcal{A} \times B$ is a finite set of links satisfying: for every $\langle i, a, b\rangle \in L$, $a \in I_i$, and for every $\langle i_1, a_1, b_1\rangle$ and $\langle i_2, a_2, b_2\rangle \in L$, if $i_1 = i_2$ and $a_1 = a_2$, then $b_1 = b_2$.

  We use Net to refer to the set of all networks.

The following auxiliary notations on networks will be useful in the sequel. Let $\mathcal{M} = \langle \overline{N}, B, L\rangle$, where $\overline{N} = \langle\langle S_1, I_1\rangle, \ldots, \langle S_n, I_n\rangle\rangle$, be a network.

- $C(\mathcal{M}) = \{1, \ldots, n\}$ is the set of *component names* in $\mathcal{M}$.
- If $i \in C(\mathcal{M})$ then $S(\mathcal{M}, i) = S_i$ is the i^{th} subsystem and $I(\mathcal{M}, i) = I_i$ is the i^{th} interface.
- $B(\mathcal{M}) = B$ is the set of buses in $\mathcal{M}$.
- $L(\mathcal{M}) = L$ is the set of links in $\mathcal{M}$.
- Let S be a system, and let $i \in C(\mathcal{M})$. Then $\mathcal{M}[i := S]$ is the network obtained by replacing $S(\mathcal{M}, i)$ with S.

We now present the semantics of GCCS systems in the standard SOS style by giving a collection inference rules to define the transitions available to processes and networks. The resulting operational semantics essentially allows one to compile GCCS specifications into labeled transition systems. Each rule has one or more premises, a conclusion, and possible side conditions. The basic structure of a rule is as follows:

$$\frac{\text{Premise}}{\text{Conclusion}}[\text{Side Condition}]$$

There are five rules. The first determines the behavior of processes, while the rest are for networks.

1.

$$\frac{q \xrightarrow{l}_Q q'}{\langle Q, \rightarrow_Q, q_0, q\rangle \xrightarrow{l} \langle Q, \rightarrow_Q, q_0, q'\rangle}$$

This rule states that process transitions are also system transitions.

2.

$$\frac{S(\mathcal{M}, i) \xrightarrow{\tau} S'}{\mathcal{M} \xrightarrow{\tau} \mathcal{M}[i := S']}$$

This rule states that a network can execute a τ-transition if one of its subsystems can.

3.

$$\frac{S(\mathcal{M}, i) \xrightarrow{l} S'}{\mathcal{M} \xrightarrow{l} \mathcal{M}[i := S']} \begin{bmatrix} port(l) \in I(\mathcal{M}, i); \forall i \in C(\mathcal{M}), b \in B(\mathcal{M}), \\ \langle i, port(l), b\rangle \notin L(\mathcal{M}) \end{bmatrix}$$

This rule states that a network can perform an input or output action provided that (1) one of its subsystems can perform such an action, (2) the action involves a port on the subsystem's interface, and (3) the port does not contain links to any buses.

4.

$$\frac{S(\mathcal{M}, i) \xrightarrow{l_1} S'}{\mathcal{M} \xrightarrow{l_2} \mathcal{M}[i := S']}[port(l_1) \in I(\mathcal{M}, i), \langle i, port(l_1), port(l_2)\rangle \in L(\mathcal{M})]$$

This rule states that a network can perform a relabeled input or output action l_2 if one of its subsystems can perform the action l_1 involving a port on its interface that is linked to a bus labeled by $port(l_2)$. Note that this rule implies that the bus in question cannot be internal (i.e. cannot be named "ξ"), since ξ is not in the range of $port$.

5.

$$\frac{S(\mathcal{M}, i) \xrightarrow{l_1} S'_i, S(\mathcal{M}, j) \xrightarrow{l_2} S'_j}{\mathcal{M} \xrightarrow{\tau} \mathcal{M}[i := S'_i][j := S'_j]} \begin{bmatrix} i \neq j; port(l_1) \in I(\mathcal{M}, i); port(l_2) \in I(\mathcal{M}, j); \\ \{l_1, l_2\} \cap InputActions \neq \emptyset; \\ \{l_1, l_2\} \cap OutputActions \neq \emptyset; \\ \exists b \in B(\mathcal{M}).\langle i, port(l_1), b\rangle \in L(\mathcal{M}), and \\ \langle j, port(l_2), b\rangle \in L(\mathcal{M}) \end{bmatrix}$$

This rule describes a synchronization between two subsystems connected to a common bus b. For such a synchronization to occur, the actions involved must come from different subsystems, with one being an input and the other an output. They must also involve ports connected to the same bus.

Some comments about internal buses are in order. As mentioned above, Rule 4 can never be applied when the bus to which a port is connected is internal. Thus, internal buses provide a means of "forcing" synchronization: the only transitions involving an internal bus that can be inferred require the synchronization of two processes (Rule 5).

4 VPL

VPL is a textual language for specifying concurrent systems. Explicit linguistic support is provided for value passing and various control and data structures. VPL-supported data types include integers of limited range as well as arrays and records.

VPL is one of the design notations currently incorporated into the GCCS coordination framework of the Concurrency Factory. As such, it can be used to specify leaf nodes in a GCCS system hierarchy. As in GCCS, however, a VPL specification is a tree-like hierarchy of subsystems, and a system is either a network or a process. So, a VPL program is only leaf-like in its position within the coordination hierarchy; in all other regards, it possesses full-fledged hierarchical structure.

A VPL network consists of a collection of systems running in parallel and communicating with each other through typed channels. VPL processes are at the leaves of the hierarchy. Each system, whether process or network, consists of a header, local declarations, and a body. The header specifies a unique name for the system and a list of "formal channels" (by analogy with formal parameters of procedures in programming languages). The names of the formal channels of a system can be used in the body of the system and represent events visible to an external observer.

Declarations local to a network include specifications of the subsystems of the network and channels for communication between subsystems that are to be hidden from the outside world. The body of a network is a parallel composition of its subsystems. A subsystem declared within a network can be used arbitrarily many times; each time a new copy of the subsystem is instantiated with actual channels substituted for the formal ones. Actual channels must match the formal ones introduced in the header and must be declared either as local channels or formal channels of the network immediately containing the subsystem.

Declarations local to a process consist of variable and procedure declarations. Procedure bodies, like process bodies, are sequences of statements. The basic statements of VPL are assignments of arithmetic or boolean expressions to variables, and input/output operations on channels. Complex statements include sequential composition, `if-then-else`, `while-do`, and nondeterministic choice in the form of the `select` statement. VPL's structural operational semantics is defined in [Tiw97] in a manner similar to the definition given above for GCCS. This semantics allows VPL system definitions to be treated semantically as labeled transition systems consisting of states and transitions, and therefore to be viewed as interchangeable at the abstract level with GCCS processes.

In the "pure" version of GCCS considered in this paper, VPL processes may not pass values to the GCCS layer. This capability is however realized in the version of GCCS implemented in the Concurrency Factory.

5 GCCS as a Coordination Language

Up to this point we have treated GCCS as a graphical description language for modeling systems that communicate synchronously. In this section we explain how GCCS in general, and more specifically the network layer of GCCS, indeed constitutes a coordination language.

Viewed as a language by itself, the network layer of GCCS provides constructs for assembling "communication architectures" with "holes" to be filled by processes. Until these holes are instantiated with processes, GCCS network structures cannot engage in execution steps; at a formal level, the semantic rules for networks all have premises that must be satisfied in order for network-level transitions to be inferred. However, once the "holes" in a network have been filled with component processes, the SOS rules for networks define how the transitions of these components may be combined into system-level transitions. Thus, at one level, the GCCS network layer enables "co-ordination structures" for GCCS processes to be constructed.

Of course, this line of reasoning does not justify viewing the network layer as a coordination language, since such languages must be capable of coordinating processes written in different notations. However, a careful analysis of the GCCS semantics, and of Rule 1 in particular, indicates that at a semantic level, the GCCS network layer does not depend on the specific syntactic form of processes beyond the syntax of action names. Consequently, *any* notation having a semantics given in terms of labeled transition systems with labels coming from the GCCS action set may be used to define entities to be used as processes. In addition, since the GCCS network constructs segregate "processes" from one another, local syntactic analyzers may be used to parse processes given in different notations, and simulators for the different notations may be yoked together inside a GCCS network simulator. We exploited these observations in our inclusion of VPL system definitions inside GCCS networks. VPL has its own independent tool support, including a compiler and a simulator. These tools are used by the GCCS coordination layer in assembling a heterogeneous system consisting of GCCS and VPL.

These observations also point to what is needed in order to include other system design notations within the GCCS coordination umbrella: the notation in question needs to have an operational semantics in terms of labeled transition systems, and the labels on the transitions need to be GCCS actions. Of course, it would in general be unrealistic to expect languages to have a semantics in terms of GCCS actions. However, existing specification languages for concurrent systems typically do have an operational semantics involving labeled transitions.

To enable components written in such languages to be coordinated via GCCS, code would only need to be written for "translating", or "marshalling", the labels into GCCS actions. Such an enterprise would involve some subtlety, since the synchronization semantics of other languages differs from GCCS. However, the effort required is much smaller than for a full-fledged language reimplementation, and it would only need to be done once per language.[1]

6 The Rether Case Study

In [DMN+97,DSC99], we applied the Concurrency Factory to Rether, a software-based real-time ethernet protocol developed at SUNY Stony Brook [CV95]. We recount this case study here in order to illustrate GCCS's role as a coordination language. Rether was designed to provide guaranteed bandwidth and deterministic, periodic network access to multimedia applications over commodity ethernet hardware. It has been implemented in the FreeBSD 2.1.0 operating system, and is now being used to support the Stony Brook Video Server, a low-cost, ethernet LAN-based server providing real-time delivery of video to end-users from the server's disk subsystem.

Rether is a contention-free token bus protocol for the datalink layer of the ISO protocol stack. It is designed to run on top of a CSMA/CD physical layer. A network running the protocol normally operates in CSMA/CD mode, transparently switching to Rether mode when one or more nodes generate requests for real-time connections. An initialization protocol is used to coordinate the switch to Rether mode. Once in Rether mode, a token is used to control access to the medium: a node can transmit data only when it has the token.

Using GCCS as the coordination framework, we specified Rether in a combination of GCCS and VPL, and verified, using model checking, two essential properties of the protocol: Rether makes good on its bandwidth guarantees to real-time nodes without exposing non-real-time nodes to the possibility of starvation. Further details of the case study can be found in [DMN+97,DSC99].

Figure 2, which contains a snapshot of the Concurrency Factory's graphical simulator in action, illustrates our GCCS/VPL encoding of Rether. It also illustrates how GCCS is utilized to coordinate systems written in completely different design notations. The center panel in the top half of the figure contains the GCCS network specification of the protocol, consisting of three subsystems, one for admissions control (`adm_ctrl`), one for bandwith monitoring (`band`), and one for the (four) LAN nodes that execute the protocol (`nodes`). Communication pathways between subsystems are formed by connecting a port of one subsystem to a port of another subsystem using a bus. For example, `adm_ctrl` communicates with `nodes` via `bus4`, which links port `release` of `adm_ctrl` with port `release` of `nodes`.

[1] It also should be noted that in value-passing GCCS, values being exchanged would also require "marshalling". Such notions may be found in other coordination languages, and in CORBA in particular.

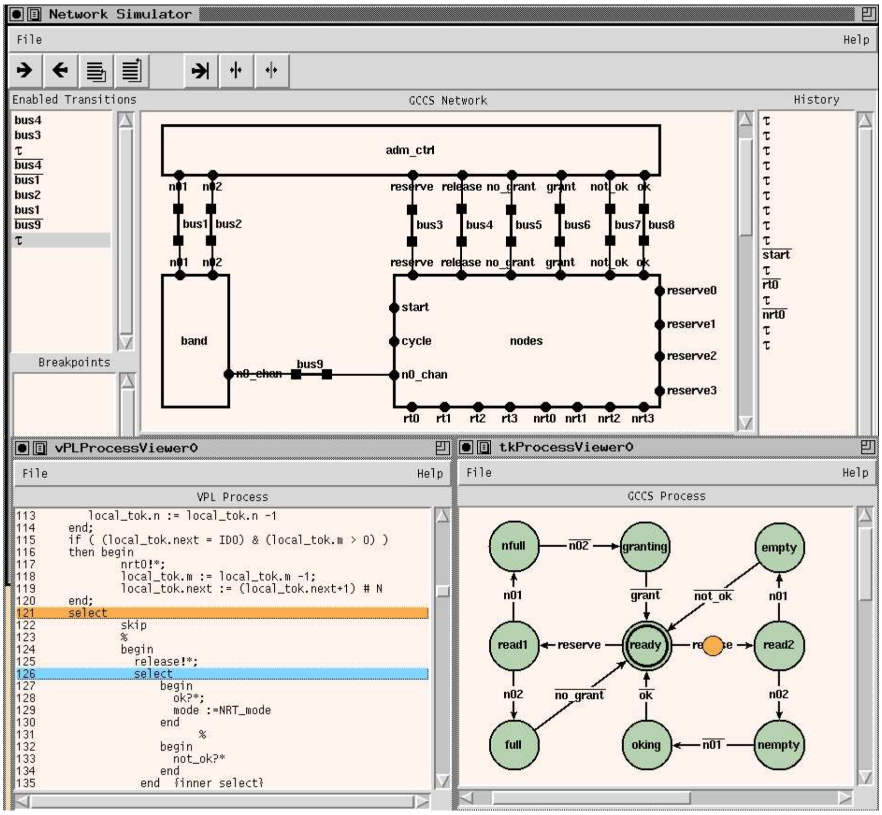

Fig. 2. A snapshot of Rether in the Concurrency Factory.

Ports in the network not connected to buses (such as `start`, `cycle`, `rt0`, and `nt0`) are for communication with the outside world, in this case, the user. The observable actions performed over these ports were used to encode the modal mu-calculus formulas we submitted to the Concurrency Factory's model checker during verification.

The other panels in the top half of Figure 2 capture some of the simulation facilities available in the Concurrency Factory, including a list of enabled transitions, a list of currently activated breakpoints (none in the example), and a history list. The list of enabled transitions indicates to the user which transitions can be executed next to continue the simulation. In this particular case, the last transition on the list (a τ-transition) is highlighted, and its execution is now being simulated. The history list shows the sequence of transitions that have been executed so far during the simulation.

The simulation menu bar is also shown (top left corner, below File). The available options, from left to right, are Go-To-Next-State of simulation, Go-To-Previous-State of simulation, Jump-Back in history list, Jump-to-Beginning of history list, Execute-Until-Breakpoint, Set-Breakpoint, and Clear-Breakpoint. The simulator's pull-down File and Help menus are also depicted.

The lower-left panel of Figure 2 is a "process viewer" tool for VPL specifications, which allows a user to scroll up or down through the VPL source code of a specification. In the illustrated case, lines 113 through 135 of the VPL source code for a LAN node executing the Rether protocol are depicted, node number 0 to be specific. Lines 121 and 126 are highlighted by the simulator to indicate that the execution of the command `release!*` in line 125 is being simulated. The effect of this command is to output a signal over port `release`.

The lower-right panel is a process viewer for GCCS processes. It enables a user to pan through a GCCS graphical process specification. In the illustrated case, the GCCS specification of the Rether protocol's admissions control process (`adm_ctrl`) is shown. The admissions control process has nine control states and twelve transitions linking control states. State `ready` is the start state.

Transitions are labeled by the actions that are executed when the transition is taken. In the current example, the simulator is simulating the execution of the `release` transition from state `ready` to state `read2`, as indicated by the small control-flow token traveling along this transition. The execution of the `release` transition along with the execution of the `release!*` command in the VPL source code of node number 0, allows the simulator to convey to the user that a communication is taking place between admissions control and node 0 so that node 0 may release its bandwidth reservation. The communication medium along which this communication occurs is `bus4`.

7 Conclusions

We have described GCCS, a graphical language for the coordination of hierarchical systems specified in a mixture of textual and graphical languages. GCCS's coordination laws are given by the set of inference rules defining its structural operational semantics.

GCCS has been implemented in the Concurrency Factory design environment using a common object interface among component languages as the basis for coordination. Currently the coordination primitives include only synchronous communication of various types of data. Future work includes expanding the set of coordination primitives to allow asynchronous communication. This can be achieved by introducing a new bus object (one that implements message queues) and extending the operational semantics of GCCS's coordination layer accordingly.

We are also interested in bringing Statecharts [Har87] into the GCCS coordination framework. Recent work involving the authors [US94b,US94a,LBC99] on process-algebraic/operational semantics for Statecharts represents a necessary first step in this direction.

Acknowledgements The authors gratefully acknowledge the contributions of Jayesh Gada, Sunil Jain, Phil Lewis, Brad Mott, Denis Roegel, Oleg Sokolsky, Pranav Tiwari, Vikas Trehan, and Shipei Zhang, to the design and implementation of GCCS and the Concurrency Factory. They would also like to thank the anonymous referee who brought references [AG97,BCD99] to their attention, and Paul Klint for discussions regarding the ToolBus [BK98]. This research was supported in part by NSF grants CCR-9505562, CCR-9705998, and DMI-9961012.

References

[AdBB+00] F. Arbab, J.W. de Bakker, M.M. Bonsangue, J.J.M.M. Rutten, A. Scutella, and G. Zavattaro. A transition system semantics for the control-driven coordination language Manifold. *Theoretical Computer Science*, 2000. To appear.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249, 1997.

[AHS93] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[BCD99] M. Bernardo, P. Ciancarini, and L. Donatiello. Performance analysis of software architectures via a process algebraic description language. Technical Report UBLCS-99-20, University of Bologna, 1999.

[BGZ98] N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, February 1998.

[BGZ00] N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In *Proceedings of AMAST 2000, Lecture Notes in Computer Science*. Springer-Verlag, May 2000.

[BK98] J. A. Bergstra and P. Klint. The discrete time ToolBus - A software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

[BZ00] N. Busi and G. Zavattaro. On the expressiveness of event notification in data-driven coordination languages. In *Proceedings of ESOP 2000, Lecture Notes in Computer Science*. Springer-Verlag, March 2000.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CGL+94] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory — practical tools for specification, simulation, verification, and implementation of concurrent systems. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Proceedings of DIMACS Workshop on Specification of Parallel Algorithms*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 75–90, Princeton, NJ, May 1994. American Mathematical Society.

[Cia96] P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300–302, June 1996.

[CLSS96a] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 398–401, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[CLSS96b] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory software development environment. In T. Margaria and B. Steffen, editors, *Proceedings of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96),* Vol. 1055 of *Lecture Notes in Computer Science.* Springer-Verlag, 1996.

[CV95] T. Chiueh and C. Venkatramani. The design, implementation and evaluation of a software-based real-time ethernet protocol. In *Proceedings of ACM SIGCOMM '95*, pages 27–37, 1995.

[DDR+99] Y. Dong, X. Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I.V. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, and D. S. Warren. Fighting livelock in the i-Protocol: A comparative study of verification tools. In *Tools and Algorithms for the Construction and Analysis of Algorithms (TACAS '99)*, Lecture Notes in Computer Science, Amsterdam, March 1999. Springer-Verlag.

[DMN+97] X. Du, K. T. McDonnel, E. Nanos, Y. S. Ramakrishna, and S. A. Smolka. Software design, specification, and verification: Lessons learned from the Rether case study. In *Proceedings of the Sixth International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, Sydney, Australia, December 1997. Springer-Verlag.

[DSC99] X. Du, S. A. Smolka, and R. Cleaveland. Local model checking and protocol analysis. *Software Tools for Technology Transfer*, 2(3):219–241, November 1999.

[DSSW99] Y. Dong, S. A. Smolka, E. Stark, and S. M. White. Practical considerations in protocol verification: The E-2C case study. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99).* IEEE Computer Society Press, 1999.

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

[LBC99] G. Lüttgen, M. von der Beeck, and R. Cleaveland. Statecharts via process algebra. In J.C.M. Baeten and S. Mauw, editors, *CONCUR '99*, volume 1664 of *Lecture Notes in Computer Science*, pages 399–414, Eindhoven, the Netherlands, August 1999. Springer-Verlag.

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

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

[S+95] M. Shaw et al. Abstractions for software architecture and tools to support them. *IEEE Trans. Software Engineering*, 21(4):314–335, April 1995.

[Sun98] Sun Microsystems, Inc. *JavaSpaces Specifications*, 1998.

[Tiw97] P. Tiwari. VPL-tool support for specification and verification of concurrent systems. Master's thesis, North Carolina State University, Raleigh, 1997.

[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.

[WMLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. TSpaces. *IBM Systems Journal*, 37(3), 1998.