Real-Time Systems, 21, 165–189, 2001

Towards a Mobile Code Management Environment for Complex, Real-Time, Distributed Systems

ALEXANDER D. STOYEN astoyen@computer.org
University of Nebraska at Omaha, USA

PLAMEN V. PETROV plamen@21csi.com
21st Century Systems, Inc., USA

Abstract. We present a novel mobile code management environment, currently under design and development. Our design employs an open architecture, suitable for "plug-and-play" with COTS and other groups' tools. While we have studied new algorithms, cost and objective functions, and other fundamental issues, the main contribution of this experimental research work is in the environment itself. It should be noted that networked platforms, such as the World Wide Web, are inherently not suitable for traditional, predictable real-time applications. Thus, real-time concerns necessarily need to be blended with others concerns, and the target applications, making use of our environment, will too be a blend of partially hard real-time and partially (or mostly) soft-real- time ones. The prototype environment will therefore support performance-based analysis and management focusing not only on predictability but also on compilation, efficiency, safety and other tradeoffs. We have selected the Java language and its bytecode format as a representation for mobile code as well as a language for our implementation.

Keywords: distributed systems, mobile code, mobile agents, compilation and interpretation, efficiency and safety tradeoffs, Java, real-time systems

1. Introduction and Motivation

In the recent years there has been quite a bit of activity and discussions on the network-centered computing concept and its enabling technologies, one of which is mobile code. The paradigm of mobile-agent computing brings profound advantages to real-time distributed computing, at a price, which seems acceptable for today's network-enabled computing resources. Mobile agents, which travel to bulky data, communicate and execute in open, heterogeneous environments, present a promising alternative to the traditional client-server approach. Agent-based applications represent more naturally the interactions in many problem domains by shifting the focus away from implementation details and towards reducing overall complexity. Mobile code overcomes many limitations of traditional approaches, caused by immobility of large or sensitive data. Thus, mobile code techniques are steadily gaining popularity not only as a research field, but also as applied technology in various industries. The hype around the network- centered computing approach is slowly fading and now we are facing the serious problem of providing mobile-code environments, robust enough to support the design and development of complex real-time systems.

In this work we examine the features and tradeoffs of such mobile code environment for complex, real-time distributed computer systems. In a modern software design and development system, support for multiple models of distributed computing is strongly desirable, but on the market, very few (close to none) products exist, which could claim

such features. Even in the research community, mobile code design and implementation tradeoffs are rarely discussed. We see a large gap between the need for sophisticated tools and integrated environments for mobile code on one hand, and the existing tools, which merely aim at enabling code mobility, on the other. Implementing mobile code, without a thorough analysis of the ramifications to functional and non- functional requirements of such mobility, could be very expensive and even dangerous for large complex systems.

In the following *Section* we outline our model for mobile code management. In *Section 3* we present related work by other groups and compare it to our approach. Next, in *Section 4* we give detailed description of the components of our tool suite and we present the challenges of our ambitious approach. We conclude with a summary of the work and additional comments on the future possibilities of mobile code in *Section 6*.

2. A Model for Mobile Code Development Environment

When designing tools and environments for modern complex computer systems, the user—a software engineer—must be provided with powerful technologies to design, develop and test systems on distributed heterogeneous platforms linked in an arbitrary topology network. This is especially true for systems, which provide code mobility capabilities. The developer needs to specify resource requirements for the mobile agents, which must be satisfied at every computational node visited. The resource allocation must be automated during the design and development via thorough analysis of objectives, specifications and matching available implementations, even in the presence of partially undefined system components. Our environment uses objective, component and platform descriptions (stated in the Unified Modeling Language—UML, (Booch, Rumbaugh, and Jacobson, 1999)) to guide the development process. System objectives are overall desired tradeoffs, which drive the instrumentation and allocation. Compilation and interpretation are not objectives per se, whereas high performance in terms of execution and communication speed is an objective (which will obviously affect the choice of allocation and instrumentation). Other objectives include security, fault tolerance, load balancing, communication minimization, and predictable timely execution. The functionality required of a component (e.g. SAR or corner turning) is similarly specified and carried through the design.

Real-Time Concerns

Mobile computing is done over networked platforms, including local and wide are networks, including the Internet. Unlike traditional embedded, tightly coupled or even distributed real-time platforms, an arbitrary networked topology is intrinsically unpredictable due to the varying delays in communications and routing, and thus such platforms cannot support guaranteed, predictable real-time computation in the sense of (Halang and Stoyenko, 1991). The question is, therefore, does it make sense to run (and thus develop and support) real-time applications over the Web and other similar networked platforms. We believe that in fact real-time applications are abundant in networked environments. However, very few

of these can be classified as hard real-time, in the sense of (Halang and Stoyenko, 1991). Rather, they are hybrid, in the sense that each has some components which have local, hard real-time constraints, while the bulk of the application is soft real-time. Consequently, predictable real-time performance is only one of a number of objectives (or requirements) of such applications. The overall objective is then formed (as already discussed earlier) by including multiple individual ones (real-time, efficiency, security and so on), and forming (combining, scaling, fusing and so on) and overall integrated objective function. In turn, this function is then used to drive such processes as component allocation, analysis and prediction of performance, routing, and so on. Among measures of real-time performance, incidentally, one may find a number of quality of service (QoS) ones next to traditional hard real-time ones (e.g., whether any deadlines at all have been missed).

A predictable real-time systems traditionalist (such as we ourselves used to be fifteen years ago) may protest this discussion and the inclusion of soft real-time into the equation. We should point out that while hard real-time controls are entirely (and perhaps solely) suitable for some applications (e.g., embedded controllers), larger- scale, complex systems are perhaps never entirely hard real-time. Moreover, deadline satisfaction may not always be the overall, sole measure of success. Consider for instance a system with two real- time processes, both periodic with deadlines at the ends of each period, with periods of, say, on the order of miliseconds and large fractions of a second, respectively. Both processes become eligible at time T_0. Which should be scheduled first? A traditional reaction would be to schedule the tighter (former) process first. Yet, we know nothing so far of the processes overall criticalities (we only know their periods). Suppose next we learn that the first process represents the processing of visible light by the human eye and the second the rate, with which heart muscles contract and pump blood. Since one can be very much alive, productive and happy, even if blind, but none of these without a beating heart, we suggest, respectively, that it may just be more prudent to schedule the "heart beat" process first. Surely then, this is both in agreement with common sense and in disagreement with traditionally-held deadline-oriented views on real-time systems.

With the key clarification of real-time concerns out of the way, we will now devote the balance of the paper to the model and the environment where hard real-time performance is but one of a number of integrated (and competing against each other) objectives.

Mobile Components and Component Relations

Mobile code components include generic classes, specialized classes, and objects. Components may have both methods and threads and depend on each other through message passing (blocking and non-blocking calls). If a callee component is passive (no threads), it acts as a classic monitor. Otherwise, its method invocations are managed as rendezvous. A generic class defines methods and generic types or constants that must be exported by any specialization or implementation, and also provides generic method parameters and attributes (e.g., QoS or security). A specialized class binds parameters and generic types to concrete types and may also state some specifics of non-functional attributes (e.g., all its implementations must provide QoS within certain bounds). An object provides a concrete implementation, including method bodies and concrete attributes. We allow no aliasing[1]

and no implicit inter-component dependencies. Mobile programs that exhibit such dependencies represent a major risk. Our tool will in fact look for such aliasing and other dependencies and flag them, as part of its operation.

Mobile Component Attributes

Components export signatures. Signatures have long been used in the formal methods community for reasoning about the interoperability of components, and for initial determination of when components are likely to be interchangeable. In addition to traditional interface and parameter information, a signature contains non-functional attributes. While system objectives represent what the developer wishes the system to be like, component attributes represent what a component *is* like, assuming some default modus operandi (e.g., the component resides at the same node as the resources it needs, is the only one requesting these resources, and is to be interpreted in its entirety). Moreover, other relevant static and dynamic information may be stated. Static information includes multilevel system description, including the componentís source programming language, targeted OS architecture, a destination node within a network, dependencies (e.g., a legacy high-performance Fortran library), and so on.

Dynamic information includes the component's current state and structure, pertinent non-functional elements, such as urgency or deadline, an estimate of execution time remaining, and so forth. We expect that components may wish to migrate or return to their node of origin for a retrofit and a reallocation. The component structure, similar to program graphs, specifies internal component organization, in terms of control flow, code segments, and calls. Importantly, the segments which form the structure are annotated with their current non-functional attributes, such as whether they have been compiled, how long they would take to compile, interpret or execute, how secure they are, and so forth. The component's current state information is correlated with the structure information.

Information on remaining time and remaining work will obviously change as a component executes, remains idle, or is passed around the network, due to the passage of time and completion of work. Less obviously, this and other information, including for example *time-to-compile*, will change as the component moves to different platforms, which have differing performance. Continuing this example, *time-to-compile* may be affected not only by specific characteristics of the compiler (possibly a different compiler), but also by more optimization, less working memory for the same compiler, or by characteristics of the platform (register set, instruction set, multiple processors, and so on). There could even be second-level effects, such as location of libraries relative to the current machine.

Dynamic information includes the component's current script. A script is a list of instructions processed by the environment's script interpreter. Every instruction requests that a particular tool (specific or generic) be applied to the component in a certain manner. For instance, an instruction may state that a commercial compiler should compile the component for a particular platform while the environment constructs a resource allocation, and then the component be launched with this resource allocation. As the requested work proceeds, the script instructions are consumed. While our environment will have a script interpreter, we do not expect this to be the case on all platforms and/or nodes. For nodes, which do not

have such a script interpreter, the scripts target the tool that hosts the component (e.g., an Internet browser). Thus, scripts are written in the language native to the host tool (e.g., for a COTS Internet browser, in JavaScript, ActiveX or VisualBasic). Scripts may include implementation dependencies, on the local compiler and platform, and dependencies on static or dynamic signature state information. While these dependencies could, in principle, be quite sophisticated, we will initially handle only a small set of simple dependencies.

Mobile Component Representation and Implementation

A component consists of three parts. The definition—in a high level language, compiled or intermediate form - defines what the object does. The likewise-stated implementation defines how the object operates (a class may have no or only a placeholder implementation). Finally, the signature provides a language- independent description of the definition, the non-functional attributes, object's source and target information, and so on. Every component originates in a certain environment and aims to execute in one or more, possibly different environments. Pertinent environmental information includes the component's implementation language, architectural dependencies, if any, in the implementation, any known message-partner specifics (e.g., upon arrival at a node, an object always registers its location at a well- known port), space, bandwidth, energy, and other non-functional attributes (interpreted as requirements for a target platform). Target platform topology, link and node capacities, resources at each node, and so forth, are also stated as requirements. This information is essential for the allocation process and very important for the tradeoff analysis that drives the instrumentation process as well.

Decisions at Local & Remote Nodes

Given a mobile code component, a target platform, and desired objectives, the environment recommends local or known remote resources to be allocated to satisfy the objectives. Given the desired QoS levels (typically in terms of performance vs. timeliness, accuracy, efficiency, security, dependability, etc.), the environment considers tradeoffs between amount of time it will take to instrument the component, compile and execute, interpret, and so on. If these tradeoffs cannot be assessed locally with sufficient accuracy, negotiation with remote nodes (where additional resources may be located) is undertaken. Given a mobile code component, whose script expects it to travel to a remote node (or, more generally, to visit a sequence of nodes), the environment recommends a route, along with necessary resource allocation, instrumentation and other decisions at intermediate points (nodes and links). Despite the seemingly different scenario, very similar questions need to be considered as for local node decisions. For instance, routing considerations are similar to those of making a remote resource available locally (or, perhaps, making the component available remotely). Similarly to the local node case, negotiation is used to evaluate tradeoffs and project eventual performance-based measures, both at intermediate nodes and at the target node, where the component is to be launched.

Delayed Access

The environment may respond to a shortage of resources (or to anomalous situations such as an active security exception) by recommending deferral of the current request until resources are likely to be available. An exception is allowed for tasks of high-priority or with urgent deadlines. Such delay will result in either rescheduling the activity at a certain time or will cause the request to be resubmitted after an appropriate interval. Similarly, the environment may instrument the component to request a resource or to re-compile itself at a later, more appropriate time.

Re-allocation & Retrofit

Given a particular allocation, which is not performing as required, the environment may be asked to recommend a re-allocation, dynamically. In turn, we may consider whether to migrate certain mobile components or the resources they use, to change the resource allocation mix, to change the component mix (by substituting relatively expensive components with less expensive, functionally-similar versions or by even removing some non-essential components), or to implement a mixture of all such changes. Similarly, given a component that is not achieving efficiency or high-assurance measures as projected, the environment may be asked to retrofit the component. That is, the component may be re-analyzed, and then re-instrumented. As part of this process, the cost and objective functions that were used to drive the original instrumentation may be adjusted as well.

Multiple Components and User Access

Given multiple, possibly interrelated mobile code components, each with a script (or perhaps a common script), the environment may be asked for local or remote, including routing, resource allocation recommendations. Moreover, the environment may be consulted by human users or agents, to speculate on resource allocation for component(s) under user control or under operation by other tools. Conversely, where allocation decisions are complex, or when there is no perfect match, and the decision is not urgent, the environment may, through our common interface, consult a user or an agent for advice. The environment may be similarly consulted on a compilation or instrumentation decision. However, given the high complexity in inter-component instrumentation, we will explore it carefully, since in this case the risks could easily far outweigh the benefits.

3. Previous Work

Our work is related to efforts in allocation and related tools, in mobile code management, and in specialized compilation/interpretation. Some of this work addresses real-time software but likely most at this stage does not. We fully expect more of this work, by us and by

others, to address real-time applications that run on networked platforms (e.g., over the WWW) in the near future.

In mobile code management, we credit Java's bytecodes (Gosling, 1995) with the first reported use of typestates (Strom and Yemini, 1986), standardized scalars, and a trusted interpreter in a major commercial language. Not surprisingly, much of the current work in this area is therefore Java- related. NewMonics, Inc.'s ongoing work was the first to aim at a clean-room real-time Java implementation (with some language additions), featuring predictable garbage collection. Many established vendors, including Hewlett-Packard, Sun Microsystems and others, now claim and offer products with the same capability. An extension of Java called Sumatra (Acharya, Ranganathan, and Saltz, 1997) features a distributed monitor that helps Sumatra programs adjust to resource availability. Support for Java agents is often provided, through research efforts (e.g., Jada (Ciancarini and Rossi, 1997)) and to some extent, commercially (e.g., ObjectSpace's 1997-announced Voyageur). These efforts do not appear so far to consider tradeoffs among non-functional objectives.

Omniware (Adl-Tabatabai, Langdale, Lucco, and Wahbe, 1996) combines the use of software fault isolation and careful high level language to detailed-level (including RISC instructions, registers) intermediate representation translation to add reasonable safety while not loosing too much efficiency in mobile code. VCODE (Engler, 1996) also uses a detailed (i.e. low-level) even if machine-independent interface (of an idealized load-store RISC machine), and very efficiently generates dynamic code. Clarity Mcode (Lewis, Deutsch, and Goldstein, 1995) is a retargetable intermediate representation for compilation on both Sun and non-Sun platforms of a simple C++ dialect (Clarity C++) developed by Sun. Auslander et alii (Auslander, Philipose, Chambers, Eggers, and Bershad, 1996) applies carefully incorporates (through templates, setups and specializations, and other methods) local optimization into dynamic code generation. These efforts appear to rely on ad hoc allocation and management.

Software architecture approaches to tools, such as the DARPA- funded Honeywell HTC work (Binns and Vestal, 1995), aim to support formal model co-generation and do some behavior prediction. However, the work does not appear to support multi-objective or mobile code systems. Two related DARPA projects at Honeywell Space & Missile Systems (with HTC cooperation) do aim to eventually produce a tool for parallelization and possibly allocation in HPC systems. However, neither mobile code nor multiple objectives have been considered. Another HTC tool (Bhatt, 1993) does perform ad hoc allocation while instrumenting low-level computational requests. Among commercial tools, Telelogic's SDT and NuThena's Foresight do perform allocation-like design, but no allocation for HPC or any parallel or distributed platforms per se. To the best of our knowledge, there are no allocation tools for mobile code components (even *conventional* program development tools for Java and JavaScript are in the emerging stages as of this writing). Some interesting research on new design paradigms for relocatable and mobile code (e.g., (Bharat and Cardelli, 1995; Baldi, Gai, and Picco, 1997; Carzaniga, Picco, and Vigna, 1997; Ghezzi and Vigna, 1997)) are emerging as well, though considerably more work will need to be done.

In addition to the work in tools and mobile code management, there are notable efforts in related research areas, including security for agents (Farmer, Guttman, and Swarup, 1996; Gray, 1996), mobile object systems and their support (Duggan, 1997; Franz, 1997), partial

information query in databases (Buneman, Davidson, and Watters, 1992; Glagowski and Jones, 1995; Shenoi, Melton, and Fan, 1992), incremental mechanisms (Hoover, 1992; Prastowo, 1995), static analysis and transformations (Marlowe and Ryder, 1990; Sreedhar, Gao, and Lee, 1995), programming environments and attribute grammars (Reps, 1988), graph algorithms, especially transitive closure and topological sorts (Prastowo, 1995; Yellin, 1993), view materialization for databases (Blakeley, Coburn, and Larson, 1989), incremental and automatic program derivation (Liu and Teitelbaum, 1995; Jones, Gomard, and Sestoft, 1993), finite differencing [PK82], and incremental languages (Yellin and Strom, 1991). A number of recent efforts are also noteworthy, trying to apply well-developed theories of program correctness to limited tradeoff considerations. Among these Necula [N97] attaches proofs or proof obligations to libraries or foreign code; verifying the proof then adds to the trustworthiness of the code. Plezbert and Cytron [PC97] analyze when or whether to compile ("just-in-time" or "better- late-than-never") code elements. Hogstedt, Carter and Ferrante [HCF97] improve predictably the parallelization of tight nested loops, which is relevant to analysis of real-time programs.

While these and many other advances are very commendable, they do not sufficiently address performance tradeoffs among mobile code objectives (e.g. compilation vs. interpretation vs. safety). Moreover, many cited systems have a strongly "pre-wired" notion of how to optimize and do not allow for flexibility nor dynamic decision-making (and unmaking). Finally, many of these systems are ad hoc (e.g., VCODE is efficient but manual and thus, in a sense, ad hoc). Our approach, through an integrated tool family, automation, and an orthogonal treatment of conflicting objectives and mobile code management, should thus be of interest and contribution the existing state of the art.

4. Details of Our Approach

We present an initial design for a powerful environment for mobile and conventional code management. The environment features an integrated set of tools for static analysis, simulation and symbolic execution, as well as run-time mobile code management. In the following subsections we outline each component of the environment and its place in the global view. Then we discuss challenges in this approach. On the diagram in Figure 1, which represents a mobile code development environment, our tools are shaded. The user interface is not shaded because visualization per se is not a primary contribution of this work, as we could develop a COTS-based UI.

4.1. Components for a Mobile Code Management Environment

Storing, Fetching, Translating, Editing

Connected and standalone mobile components are stored in commercial databases, Internet repositories and other sources (perhaps, a user develops a mobile component anew), as are descriptions of resources, the visible network topology, and so on. Search engines are used to find or place items in these databases and repositories, based on their functional

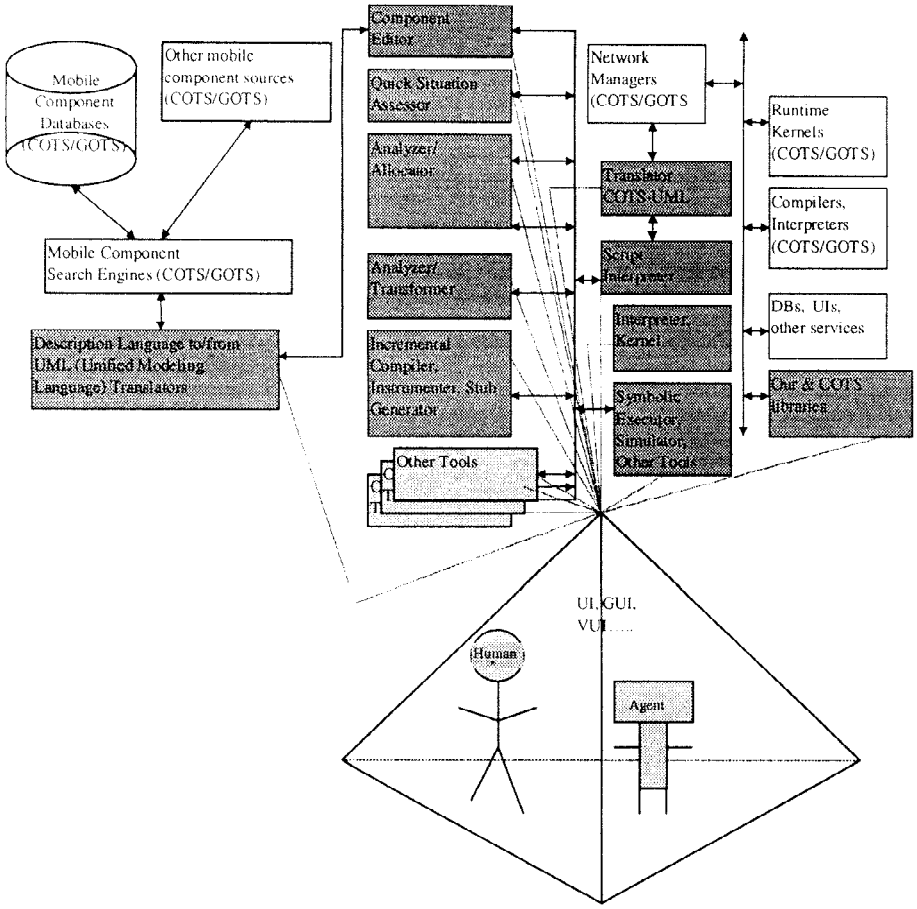

Figure 1. A mobile code development and management environment.

and non-functional descriptions. A previously processed component may also return from a remote host where it was launched, for a retrofit. Component descriptions are updated by the Component Editor, which is also responsible for extracting and updating multilevel resource and network descriptions. Translators are used to map other COTS/GOTS description languages to and from UML, which the environment uses for representations. Formal techniques will prove useful in deriving such translators and will reduce the amount of effort in providing translators for different platforms and notations. A component or set of components (or, in some cases, alternatives for components) is proposed, built and/or fetched, typically on the basis of system objectives. Resources are selected and fetched (not necessarily in that order) from local information and/or resource repositories, using tools described below, on the basis of system objectives, signatures of services requested

and provided, and such system state information as is available. Through the Component Editor the Translators are instructed to map the objectives to appropriate search engine query structures. In addition to components and resources, the Editor also works on system objectives, component attributes and scripts. The edited (or simply passed through the Editor) components are worked on by other tools, including Analyzer, Allocator, Transformer, Compiler, Instrumenter, and others. As discussed below, which tool works on which component(s) and when is governed by scripts, created or modified, in turn, by human users and/or agents.

Deciding, Analyzing, Transforming, Instrumenting, Allocating, Compiling, Generating Stubs

Once a mobile component (or a set of communicating components) has been made available, a Quick Situation Assessor is tasked to look at the component first. The Assessor is most valuable when a component needs to be sent somewhere in a hurry and in turn requires some basic resources, quickly. Similarly, the component may need to substitute for a failed component or it may require an urgent retrofit. The Assessor's job is to determine quickly what if anything (e.g., compilation, instrumentation) needs to be done to the component and/or whether it can be provided with the needed resources and sent according to a quickly-selected route to (a reasonable) destination. Alternatively, the component may be referred to the Analyzer/Allocator, forwarded to the Script Interpreter (with an updated script), returned to the Editor, or declared a failure (e.g., if the Assessor determines that the component will necessarily miss its deadline by the time it may possibly get the needed resources). If the component is not suitable, given the objectives, the Assessor may request that the Editor fetch a less expensive, functionally- similar component version, and that the human user or system architect be informed. Formal methods have proven to be useful in promoting software reuse, and in particular in identifying functional compatible components. The principle of step-simulation and refinement enables the identification of components that although not necessarily 100% compatible, can be used to implement particular patterns of behavior. Such techniques will be employed where their use is expected to reduce costs in determining appropriate matches.

Components chosen for detailed analysis and possible retrofit are worked on by the Analyzer/Transformer. This tool examines the component vis-à-vis a certain tradeoff among conflicting objectives, and applies program transformations to achieve optimality on the tradeoff. Examples of transformations include replication for additional fault-tolerance, removing an unnecessary security check or array subscript range check, or changing a callee-method binding to a less precise but faster version, to speed up execution. In the last case, the replacement might be temporary, providing additional time and resources for optimization of this or another task, allowing rescheduling on a change in task set, or otherwise responding to a change in a requirement or in the underlying distributed platform. Components not requiring any more analysis or transformations are (incrementally) compiled and instrumented, as required, to force satisfaction of non-functional objectives as well as to self-report, self-monitor and fit in host environments. Components chosen for detailed analysis and allocation are assigned by the Analyzer/Allocator. This tool exam-

ines the component (or multiple components) and its resource requirements, which may be conflicting among each other, and certainly are competing with other components, against available system resources. The examination is performed vis-à-vis a particular objective function (which drives the allocation) and following a resource allocation heuristic.

For both tools, both the function and the heuristic may be switched until a suitable allocation/instrumentation choice is found or the tool gives up. For a suitable set of resources to be found, the set must conform to the requirements, profess sufficient quantities available (given its current and projected future use), and be located in such a manner that a suitable strategy for making the resources available to the component can be constructed. In a high- performance environment, suitable strategies are expected to include code and data movement and re-allocation, to optimize the overall access. In particular, avoiding massive computation and communication imbalances is a frequently expected necessity (even though the components are expected to request massive amounts of computation and communication). If successful, the tool generates an allocation and a strategy for implementing it. The component's script is updated accordingly and the Script Interpreter is asked to take over, to implement the allocation. For instance, the script may identify a set of five resources, needed by the component, describe a route to get the component to a node with the three of the five resources, order that the fourth resource be sent to that node, and generate a stub to travel to the node of the fifth (bulky) resource, to execute there on behalf of the component, returning the results to the component.

Components that require stubs to be generated are passed to the Stub Generator. Again, refinement techniques generated in the formal methods community will prove useful in this process, generating stubs to match particular requirements identified earlier. Stubs may be needed because a component may require the use of a distributed set of resources, necessitating in turn remote use of some, and thus stubs for communication. The component may also request (through instrumented "triggers") that it be (incrementally or partially) compiled at a remote host. Stubs support the passing of arbitrary parameters (including methods and objects), call-backs and synchronization, and optimization of call and return data streams, to facilitate high-performance, high-confidence computation and communication needs. (Un)marshaling and conversion of parameter messages suitable for heterogeneous (in hardware, OS, and languages) environments are performed here as well.

Interpreting, Executing

The Interpreter interprets source or intermediate form, and the Kernel executes compiled components. In principle, if the code is properly instrumented it may run on a COTS kernel (or interpreter) and still operate correctly. For instance, the code may at a certain point request that a routine (inserted earlier by the instrumentation) be executed. The routine will assess whether the functional code to be executed next should first be compiled or just interpreted. Depending on the outcome, the routine will pass back to its caller a handle via which the next step (compilation and then execution, or straight interpretation) will be executed. However, given the widely varying degree to which COTS interpreters and kernels provide suitable APIs for such functions, it may not be straightforward to build such functions generically and then specialize them for particular COTS elements. Moreover, we

foresee potential efficiency sacrifices in implementing such functions. Finally, we are not certain that COTS kernels and interpreters (and compilers) can be easily used for incremental "just-in-time" compilation. Thus, one must be prepared to extend COTS interpreters and kernels through pre-/post-processors and API extensions.

What differentiates our Interpreter and runtime Kernel is the ability to follow instrumented requests, such as assessing the degree of satisfaction of objectives, triggering partial (re-) compilation, or sending the component back for a retrofit. Our experience in building such tools also suggests that instrumentation may conflict with general scheduling and resource allocation. For instance, a general preemptive scheduler destroys any guarantee or measure of timely performance, when the measure is triggered at arbitrary points. On the other hand, when preemption and triggers are restricted to occur only at call- and return-times, such guarantees and measures are possible. While it may be necessary to build our own Interpreter and Kernel, a better solution would be to augment (or provide suitable schedulers for) COTS ones.

Script Interpretation

Script Interpreter processes scripts, which are instructions how to allocate, execute, interpret, compile and otherwise process mobile components. Naturally, a component may arrive via the network, from a remote node or may originate at the local node. Whichever node launches the component, the Script Interpreter gets it after the component has been built, analyzed, transformed, compiled and is ready to be processed, according to its script. The script may in turn ask that the component undergo a re-allocation, or additional compilation, or analysis, or other steps, at a later stage.

COTS Environments and Tools

In addition to our tools, the environment interfaces with various COTS packages: network managers, kernels, interpreters, compilers, and various utilities, available on the network. Our approach is to produce an open environment, one adhering to relevant emerging standards and interoperating with conventional tools, other mobile code tools, and various services and functions. Clearly, we cannot expect a conventional tool to provide the same level of detailed service for mobile components as our environment does. However, we can expect that a reduced, "conventional" level of service be provided. For instance, a COTS Internet browser will allow for manual allocation (without any concomitant analysis or stub generation, other than that provided by the native language environment). Similarly, a COTS browser should be able to interpret the entire component (ignoring instrumented directives to compile and execute some parts). Moreover, COTS browsers and other tools should also allow for user defined "escapes" which can be used in turn to invoke the tools that we provide.

Libraries, Auxiliary Functions

While mobile components likely reside in databases, some auxiliary objects will be stored as library entries. For instance, a COTS component may be represented using a proprietary IDL. Then, translation routines to and from UML have to be provided, as library entries. Similarly, we envision storing wrappers, RPC (un)marshaling and conversion templates and routines, and others. Description and interfaces of these objects are exported. Again, formal techniques are expected to prove to be useful in finding matches.

Atomic cost functions, scaling, fusion and conversion strategies, and example functions (cost and objective) are stored. The descriptions of all these objects are exported. Some functions will be marked for use in aggregates and others on their own. Related to these functions, heuristics to integrate and evaluate them too are provided and stored in libraries. While space considerations prevent us from presenting a detailed discussion of allocation in this paper, we will put our considerable expertise in this area to extensive use here. Essentially, a system consists of mobile components, which require resources, with functional and non-functional properties. The goal is to allocate the resources, subject to the conflicting objectives, constraints, and in the best possible way (there is significant competition for the resources). This allocation process differs in level of detail only, across different stages of the system lifecycle. Given the general growth of the problem space the process has to consider, the Allocator must make use of heuristics and approximate "best effort" strategies (exact allocation is computationally prohibitive). Thus, we will thus store a slate of heuristics and their descriptions in libraries. These will include commonly available and "home-grown": genetic, neural net, projection, greedy, and others. Three modi operandi will be supported: search (entire space is considered per iteration), construction (a fixed number of objects are considered per iteration), and hybrid (the heuristic relies on a strategy/user to switch modes; e.g. first, do a quick global search, then optimize-construct a local region. . .).

We store routines that perform basic approximate and imprecise (converging and improving over time) arithmetic and symbolic computation. These are used as the need arises in the environment's activities (e.g., as a mobile component is built, instrumented, and allocated). Owing to the special needs of embedded computation, we anticipate storing pre-designed low power (i.e., a computation using power-inexpensive software and hardware implementations) versions of common computations (such as the Fast Fourier Transform, used heavily in real-time imaging). While we should not need to develop new analyses or transformations (or as few as possible), existing analyses and transformations will have to be translated, and in some cases somewhat modified, to operate within our framework and environment. These analyses and transformations are then stored as components. In fact, since there may be competing analyses with different resource tradeoffs, or even providing different levels of precision at different costs, and likewise competing transformation engines or transformation orders, these analysis and transformation components are accessible by component matchers in a manner identical to other components.

Selecting, analyzing and allocating mobile components is inherently a heuristic process. A componentís script dictates how to manage the component, stating when to allocate, launch, run, and so on. Standard scripts are stored in libraries, and component-specific

scripts are built, on the basis of component construction decisions and heuristic knowledge, from these scripts. Similarly, agent scripts are developed and stored in libraries as well. Other libraries will include compiler, OS, GUI, DB and other utility libraries.

Symbolic Executor, Simulator, Evaluator

To support detailed performance and other assessment for instrumentation, compilation vs. interpretation vs. safety vs. performance tradeoff analysis, and allocation, we provide a symbolic executor, simulator, and evaluator. These tools symbolically execute, simulate or otherwise evaluate the would-be compiled, instrumented or allocated high- performance mobile code on target platforms. Simulation and symbolic execution are more expensive but also more detailed substitutes of objective function evaluation. When in the loop, the human user interacts with these tools and with the simulations. Important concerns, such as resource contention, identified in the simulations, are reported using colors, shapes and other props. Moreover, the user is able to change key variables and trying alternate scenarios. We envision eventually supporting four types of system sources, listed in the increasing degree of detail (and time- space cost of use):

1. A mathematical system description (e.g., a system of constrained iterating equations); using this method, it is possible to assess certain component-level non-functional objective satisfaction (e.g., component reliability) but only as boundary cases; it is impossible to assess functional or behavioral characteristics.

2. A system description consisting of object interfaces (functional and non-functional information); using this method, it is possible to assess both component-level and certain inter-component non- functional and some functional (e.g. amount of inter-component communication) objective satisfaction.

3. A system description consisting of object interfaces and flow- skeletons (calls, conditionals, iterators but no actual functional code); cycle-burners and resource-idlers written in the native language are used instead of the functional code, approximating closely the codeís non-functional aspects; using this method, it is possible to assess both intra- and inter-component non-functional objective satisfaction.

4. A system implementation; in principle, everything fundamentally assessable can be assessed in this representation.

UI, Control, Humans, Agents

The environment uses an intuitive and powerful User Interface (UI). Operations on mobile components, both during runtimes and in-between runtimes can be both monitored and accessed. The control of all tools is accessible to the user, though every tool may operate automatically as well (in which case the user may monitor and, if need be, interact with the operation). Multiple mobile components may be of interest at any given time, some worked on locally, others executing locally, others worked on remotely (by our or other environments

on remote nodes), and yet others executing remotely. In parallel, different tools or copies of tools of the environment may be operating, at different stages of progress. For instance, one mobile component may be under construction, another undergoing an allocation, another in the process of being launched, another undergoing a retrofit/re-instrumentation, and yet another executing. All such actions are displayed using an intuitive, graphical and, where feasible, visual representation, including colors, shapes, icons, sounds, and animation. The environment is used by humans and artificial agents. A human user is typically a programmer, a network administrator, or possibly an application specialist. An agent is a program with its own thread of control and a function, which uses the tools, and assists with instrumenting, resource selection, route plotting so on. For instance, the Assessor may have decided that there was no time for detailed analysis of security and performance tradeoffs to drive the allocation, and that a mobile component be allocated default resources and sent to a default node. In parallel, an agent may perform various tradeoff analyses and determine that a different resource selection may be more appropriate, delivering reduced but still adequate security but much better performance. The agent may then request (with or without human concurrence, depending on the circumstances) that its component allocation will be the one used and the original user selection should be "undone" and terminated.

4.2. Hard Issues and Supporting Technologies

Component, Platform and Objective Descriptions

Inference of values for component, platform and objective descriptions is a hard fundamental issue. We will address it by restricting resource- algebraic and timing expressions to subsets for which automated proofs exist, and/or for which static analysis, partial evaluation and specialization are possible. For cases which resist such analyses, we will allow symbolic execution and simulation verification, for as long as the number of cases to consider is kept small. Otherwise, we will force approximate and imprecise solutions of the expressions. Before any approximations are done, however, the user will be informed and queried for possible re-formulation. Our tools will also suggest expression alternatives to the user (through agents).

Another challenge will be recognition of new resources and resources at previously unseen sites. If the remote site is using our or compatible tools, a resource description will be available, and it should in most cases be reasonably simple to check the validity of the descriptor, at least relative to the services and level of security required. Protecting against malice, or at the extreme levels of the service or security demand, will admittedly be more challenging. Otherwise, the tool will have to rely on a library of resource components, and a resource-recognition algorithm. We intend to rely on commercial databases, possibly with filters, to provide the library and its search engine; the recognition algorithm is a more complicated issue, which will be addressed during a later phase of this work.

Some of the needed methods and analyses already exist to an extent in the CRL platform (Stoyenko, Marlowe, and Younis, 1995; Stoyen, Marlowe, Younis, and Petrov, 1997), and in the Destination resource allocator (Marlowe, Stoyenko, Laplante, Daita, Amaro, Nguyen, and Howell, 1996; Amaro, Marlowe, and Stoyenko, 1996; Harelick, Marlowe, Stoyenko,

and Sinha, 1995; Stoyenko, Welch, Laplante, Marlowe, Amaro, Cheng, Ganesh, Harelick, Jin, Younis, and Yu, 1993). Others are available from other groups, in the literature or are under development. We also have substantial expertise in formal method approaches to automated proofs, incremental verification and validation, and related techniques (Bowen and Hinchey, 1994; Bowen and Hinchey, 1995a; Bowen and Hinchey, 1995b, Farrow, Marlowe, and Yellin, 1992; Halang and Stoyenko, 1991; Hinchey and Jarvis, 1995; Hinchey and Bowen, 1997; Hinchey and Bowen, 1995; Marlowe and Ryder, 1991, Marlowe et al., 1993; Masticola, Marlowe, and Ryder, 1995; Silberman and Marlowe, 1996; Younis, Marlowe, and Stoyenko, 1994; Younis, Tsai, Marlowe, and Stoyenko, 1995). Thus, while the problems of value inference and determination are very hard, they can and will be addressed sufficiently to facilitate the environment construction.

Cost and Objective Functions

Our tools capture both requirements and promises as cost functions. These are fused, scaled and converted into an integrated objective function, which is, in turn, used to measure the "fitness" of a component or of an allocation. We have thoroughly researched cost and objective functions. In the overwhelming majority of classic efforts (Callahan and Purtilo, 1991; Houstis, 1990; Levi, Mosse, and Agrawala, 1988; Hoang, 1991; Lo, 1988; Stone, 1977), cost functions are represented as constants or scalar variables. The objective function is consequently computed as a linear expected-value summation, with either constant or constant- sum weights, representing the significance of each cost function's contribution. However, such approaches may fail to represent adequately the situation, for a number of reasons, including following: (1) significance values may change over time, (2) individual objectives may exhibit a dependency on each other, (3) the integrated objective relationship may not be linear. Consequently, and on the basis of our extensive experience in tools and languages (Stoyenko, 1987; Stoyenko and Georgiadis, 1992; Marlowe, Stoyenko, Masticola, and Welch, 1994; Kligerman and Stoyenko, 1986; Younis et al., 1994; Younis et al., 1995; Laplante, Marlowe, and Stoyenko, 1996; Amaro, Marlowe, and Stoyenko, 1996; Wei, Stoyenko, and Goldszmidt, 1991; Halang and Stoyenko, 1991; Harelick et al., 1995), we have developed both a detailed hierarchy (with multiple inheritance, e.g. a latency cost function derives attributes from both real-time performance and fault- tolerance objectives) and a general equational form for an objective function, derivable recursively on the basis of the hierarchy (Real-Time Computing Lab, 1996). In brief, a function is represented recursively as a function of more detailed cost functions, divergence, fusion, scaling functions, and various algebraic combinators. Default definitions of these functions correspond well to simple, common objectives. Numerical method solvers are easily applied to this form and thus, the form can be used in our tools.

As expected, the equational form supports conversion, scaling and fusion of cost function information. While conversion and fusion are typically found in discussion of cost functions, scaling is also very important. A classic example of use of scaling is in a HPCC system where the network is much slower than the general-purpose computers. Consequently, to avoid an integrated objective function which is very sensitive to network cost changes and very insensitive to computation cost changes, we may wish to express network timing as

slower units (e.g., in 10E-2 sec) and computation timing in faster units (e.g., in 10E-4 sec). While normally an assessment step is evaluated by computing the objective function (a fast, closed-form operation), the environment also provides more detailed, but time- consuming tools e.g., symbolic executor, and simulator. It is expected that the use of these detailed tools, will provide feedback on both the allocation quality and on that of the objective function itself. Thus for instance, a simulation may reveal that a component generates excessive communication traffic, while the communication minimization cost function is seemingly well- represented in the objective function. For this reason, the objective function may be changed, dynamically, by the user (acting upon the environment's advice), or even automatically (e.g., the communication minimization term may be multiplied by a handicap factor).

Compilation and Interpretation

Compilation and interpretation are two alternatives for the translation of a component to platform-specific code and subsequent execution of that code ñ or more accurately, two ends of a spectrum of approaches to translation and execution. Full compilation translates the entire source-language program into an intermediate form, in a (largely) platform- independent manner, followed by language-independent translation of the intermediate code into platform code and scheduling of that code, followed in turn by execution of the platform-specific code. The key point is that the entire program is translated before execution. In some cases, particularly with explicit language parallelism, or for real-time and similar applications, the translation to intermediate form has to have some awareness of platform properties. Interpretation relies on a statement-by-statement, as- needed translation of source code into machine code, followed by the execution of that statement, together with updates to the program environment (essentially, the symbol table and store). Compilation requires lead time and resources for its initial phases, but results in faster code and potentially better resource usage, both because the translation overhead can be avoided, and because translation has some global (although static) knowledge. In addition, compilation allows more time for analysis and transformation to improve the resulting code. In contrast, interpretation has higher per-statement execution cost, but can begin to execute immediately. Interpreted code is more robust, because the platform-code generator has current and dynamic knowledge of types, values, and so on; in some cases, such knowledge may even lead to more efficient resource usage. However, only source-to-source transformations, and some local optimizations, are easily realizable.

There are a number of intermediate alternatives. For example, components may be stored in intermediate form (translated by the front end of the compiler). Carrying this process somewhat further, Java and related languages compile to an intermediate form (byte code for the Java VM), which is then instantiated for the particular platform and interpreted. In either case, interpretation cost is significantly smaller, and a fair amount of analysis and transformation is possible, although resource usage tends to correspond to the interpreted case. The other approach is to compile as needed. It is possible, the first time a statement is interpreted, to save a template for the code to be generated, which again cuts down on interpretation time if the statement is revisited. It is also possible to compile in

the background for subsequent execution, once we know that a component is to be used on a particular platform. Finally, various groups are exploring dynamic or just-in-time compilation, in which code is compiled as seen.

Compilation and interpretation also interact with instrumentation, stub generation and linking. A component pre-compiled, even as far as intermediate form, can have hooks inserted for instrumentation and for stubs. At a minimum, the Instrumenter and Stub Generator can transform these into more fully elaborated hooks, using in addition UML annotations, plus symbol table and dependency information provided by the compiler or by analysis tools. Some will be independent of both platform and caller/callee/message partner, and can be fully expanded immediately; most however require either knowledge of the target platform, or specifics of the interacting process(es), and so can be resolved only when a set of components is linked (either on a particular platform or into a platform- independent hierarchical component), or when the component is instantiated on a particular platform; determining efficient staging and correct binding times for various pieces of information needed for these tools poses an interesting problem. It is more difficult to provide such support in a pure interpreted scheme: stub generation and instrumentation both benefit from global knowledge, and in the former case, even coordination between components, at least at the level of signatures. This information will not in general be available to the interpreter when stubs or instrumentation are to be generated. Hybrid schemes do not, however, appear to suffer from the same difficulties, since compilation will typically have proceeded far enough to make signatures and most other global information visible.

Capture of Transformation Rationale

Another challenging issue is the systematic capture of the assumptions that guided various transformations. For example, optimization for average-case execution time may produce poor code for real-time applications, may inhibit compiler extraction of parallelism, may lead to resource contention in resource-poor environments, and so on. Another example is the notorious sensitivity of behavior of parallelized code on platform characteristics, and the need to take account of the resources and types of parallelism provided (and not provided) by the platform. A third and important example is the sensitivity of optimization of called routines on the aliases which are visible therein; as mentioned above, careful design can reduce but not eliminate the impact of such aliasing. To emphasize the need for capture of transformation assumptions, note that classical compiler optimization, and a number of other transformation and analysis techniques, are directed at improving a single aspect, or a few standard aspects, such as average-case performance, minimization of communication, and memory usage. However, most of these transformations do not respect other constraints and objectives, such as power consumption or heat generated, worst-case performance (important for real-time applications), predictability, security, fault-tolerance, or portability (although individual cases of course differ). Under various combinations of criteria and transformations, we may be able to use the original transformation, use a guarded or modified version, use another transformation ordinarily not as preferable, or not be able to transform at all. While we do not believe a generic remedy or even classification scheme is possible, we will continue to investigate this issue and fully expect to address a number of concrete and

practical situations (e.g., timely performance versus minimization of memory usage), where we already have a good understanding of the tradeoffs, in our prototyped environment. A mobile code component may thus exist in one or more of a number of forms, including source code, intermediate code, virtual-machine code, library target-code templates, code specific to a family of platforms. It may have versions, optimized or transformed under particular assumptions. Finally, it may be translated and executed through a variety of schemes, several of which have been outlined above.

Stub Generation for High-Performance, High-Assurance Computation and Communication

While stub generation is not a new challenge per se, stub generation technology suitable for high-assurance HPCC software is not commonly used or understood. Stub generation needs to support distributed, networked and heterogeneous resource use by mobile components. The use must not be in the way of high performance and other objectives, while massive computational and communication loads are to be expected. To support heterogeneous resources, we employ template-driven stub generation as we did in (Wei et al., 1991). A template is provided for every kind of stub (declaration, call, accept, returnÖ.) and for every basic type or type constructor for parameter passing. A set of templates is provided per language supported, in the native format, and an associated UML set is provided to enable our tools to manage the templates. To support massive loads, we generate stubs suitable for RPC and provide an RPC management system based on our Inverse Remote Procedure Call concept (Stoyenko, Bosch, Aksit, and Marlowe, 1996). Essentially, the stubs are instrumented to trigger either data to code or code to data flow. The IRPC system considers the current system performance (global or local) and decides how to split the computation into local and remote parts and what flows to provide for. Once this decision is made, the Allocator is presented with a selection of allocatable components (which are pieces of bigger components, split along the flows, if need be replicated partially at remote nodes, and augmented with stubs). To support stub distribution as well as the invocation of remote stubs for inverse RPC flows (code to data is considered ìinverseî, data to code ìdirectî), general subprogram passing as parameters on RPC calls is implemented, as in SUPRA-RPC (Stoyenko, 1994) (this includes support for out-of-scope side-effects if any). Finally, should we discover a need to interoperate with any COTS RPC or ORB management systems, we will build an RPC-level interoperable model, as we did in (Stoyenko, 1991).

Safety

A primary impetus for the development of Java and related languages and their environments has been the need for safe execution of code. From our tools' perspective, unsafe execution occurs when a request is given to a resource which cannot be trusted to handle it, or when a request has the potential of corrupting the resource. Java provides protections, although not absolute guarantees, against the latter, via its Security Manager. Our tools will likely adopt a similar strategy for remote requests. We may also develop analyses by which to certify foreign components and/or their requests as safe for particular resources, as well as

filters through which requests can be safely processed while bypassing resource-intensive access to the full Security Manager. The former is more difficult. While local resources can presumably be trusted or at least be checked for correctness, verifying the security of remote resources is very difficult. Where the application does not need an immediate response, we can use the remote resource "on spec", and again use a filter to check the result for correct message structure (e.g., type and typestate (Strom and Yemini, 1986)), and against memory access errors. We look only to prevent disaster. We cannot check (with rare exceptions) for the correct answer. We have to trust the internal semantics the same way any component manager has to trust the semantics of components whose signature and claimed invariants match. On the other hand, it is clearly desirable to be able to have some assurance that another component or resource, even one supposedly analyzed and annotated using our tools, can be trusted. We also hope to develop a protocol whereby the Analyzer tools on the two sites can negotiate to obtain some degree of assurance of the safety of both components/requests and resources on remote sites.

For compilation-related operations, security has several dimensions, the most significant being memory access faults ("errors" if they are accidental) and file/resource/command permission violations; the former is the motivation for the strong typing and the absence of explicit pointers in Java, the latter for Java's security manager. Related to memory access faults is unrestricted aliasing; this is addressed in part again through restriction of pointers and type casting, although (unlike Ada) Java does effectively allow aliases, through essentially mandating the use of callbacks and handles to manage graphical and other components. Hermes [BSY90] allows for typestate annotations, which address other related issues by adding to type signatures information about constant values (also present in C++) and initialization/non-initialization of values; Hermes also allows one to define a lattice on the values sent or returned on a call, and to require by an annotation that the value of a parameter to be higher (lower) than a given value, or the values of two parameters, or the IN and OUT values of one parameter, to be related. Still another related issue is value-based exceptions (such as overflow, or divide-by-zero) and their handling.

While it is clearly desirable to have secure, safe, error-free components, there are at least three arguments against requiring all components to be so certified. First, it is impossible to provide such assurances for a general-purpose language through a combination of language design and static analyses—problems such as array-index- out-of-bounds, divide-by-zero, and null-dereference cannot in general be caught by such means. Even when a guarantee is possible, it may be difficult to provide in combination with desired language characteristics; for example, apparently there are holes in the Java type system which allow memory access violations. Second, as we have seen above, a certain degree of aliasing (and likewise type casting, etc.) may be desirable or even necessary for some classes of programs. Third, and perhaps most importantly, there are significant and sometimes unacceptable compile-time and run-time costs in providing such security. For example, range analysis, which, among other applications, can show that array accesses are in bounds, can be, even when it converges, an extremely time- consuming analysis, especially interprocedurally. When a component has to come on line immediately, it should be preferable to forgo the analysis, and use dynamic checks instead. We may even choose to believe an annotation on a caller from a trusted repository, that out-of-bounds values will never be delivered, and remove

checking altogether, at least until an analysis can be performed in the background. Likewise, we may choose not to keep checking for changes in a global variable, perhaps risking some incorrect and even formally unsafe computations, if we can roll back the computation. We may choose to forgo checking for some abstruse exception by name, or not to provide a handler for it, if we have reason to believe that it would only occur through malice, and that (perhaps because all access is restricted at another level) this is unlikely.

5. Conclusion

Arguably, we have experienced the most incredible explosion in software and hardware technology in the past decade. The rapid advances in hardware and the wide availability of a spectrum of computing resources, most of which are interconnected via local networks or ultimately through the Internet, has generated considerable interest in new approaches to distributed computing. Mobile code is the new enabling technology for an array of applications from Internet search engines (web-crawlers) to medical database connectivity to telecommunications. Considerable effort with commendable results has been invested in research and development in this relatively new area. We feel now is the right time to consider these achievements in the global view of complex real-time computer systems development. Mobile code is clearly a necessity and not merely a research toy anymore and environments supporting it will become significantly more commercially available.

In this work we establish guidelines for mobile code management and define features of an integrated development environment for mobile code applications. We consider all angles of the problem of introducing code mobility to large complex system and we propose tools to automate the design and enable the analysis of such systems. Further, we provide a methodology and tools to analyze performance tradeoffs for mobile and traditional software components, in an environment of scarce resources, via global resource allocation and objective functions. For mobile components we examine performance vs. safety vs. portability tradeoffs to determine what combination of compilation and interpretation of the transmittable code satisfies best the performance objectives set by the developer. We utilize objective functions and functional and non-functional description of the system components to characterize the resources required and the performance guaranteed by the component. In a dynamic global resource allocation problem we utilize heuristics to maximize the satisfaction of composite objective functions, which represent real-time and other non-functional concerns, for the entire system against the available QoS guarantees. Thus, we provide detailed guidelines and specific insight into constructing an integrated design and development environment and tools with support for mobile code.

Notes

1. Certain sorts of applications may, however, rely on the possibility of aliasing, particularly as array section aliasing for scientific numerical codes, and or confluent access path expressions in regular dynamic structures, such as doubly-linked lists. Component annotations may include such permitted, and even required, aliasing patterns.

References

The programming language Ada reference manual. 1983. ANSI/MIL-STD-1815A, LNCS 155, Springer-Verlag.

Adl-Tabatabai, A.-R., Langdale, G., Lucco, S., and Wahbe, R. Efficient and language-independent mobile programs. *PLDI'96*, pp. 127–136.

Amaro, C., Marlowe, T. J., and Stoyenko, A. D. 1996. An Approach to Constructing complex evolving systems using composition of knowledge domains. *21st IFAC/IFIP Workshop on Real-Time Programming*.

Auslander, J., Philipose, M., Chambers, C., Eggers,S. J., and Bershad, B. N. 1996. Fast, effective dynamic compilation. *PLDI'96*, pp. 149–159.

Acharya, A., Ranganathan, M., and Saltz, J. 1997. Sumatra: A language for resource-aware mobile programs. *Mobile Object Systems: Towards the Programmable Internet.* Springer-Verlag, Lecture Notes in Computer Science, No. 1222, pp. 111–130

Bhatt, D. 1993–96. SPIE, Honeywell Technology Center, Minneapolis, MN, USA.

Bharat, K. A., and Cardelli, L. 1995. Migratory applications. *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology.* Pittsburgh, PA (also available as Digital Systems Research Center Research Report 138).

Binns, P., and Vestal, S. 1995. Architecture specifications for complex real-time dependable systems. *First IEEE International Conference on Engineering of Complex Computer Systems.* Ft. Lauderdale, FL, USA, pp. 357–360.

Blakeley, C. J., Coburn, J., and Larson, P. 1989. Updated derived relations: Detecting irrelevant and autonomous computable updates. *ACM Transactions on Database Systems* 14: 369–400.

Brodie, L. 1981. *Starting Forth*. Prentice-Hall.

Baldi, M., Gai, S., and Picco, G. P. 1997. Exploiting code mobility in decentralized and flexible network management. *Proceedings of the First International Workshop on Mobile Agents.* Berlin, Germany.

Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The Unified Modeling Language User Guide*. Reading, MA: Addison Wesley.

Bowen, J. P., and Hinchey, M. G. 1994. Formal methods and safety critical standards. *Computer* 67–71.

Buneman, P., Davidson, S., and Watters, A. 1992. A semantics for Complex objects and approximate queries. Department of Computer Science, University of Pennsylvania, Philadelphia.

Callahan, J., and Purtilo, J. 1991. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering* 17: 626–635.

Carzaniga, A., Picco, G. P., and Vigna, G. 1997. Designing distributed applications with a mobile code paradigm. *Proceedings of the 19th International Conference on Software Engineering.* Boston, MA.

Ciancarini, P., and Rossi, D. 1997. Jada—coordination and communication for Java agents. In *Mobile Object Systems: Towards the Programmable Internet.* Springer-Verlag, Lecture Notes in Computer Science, no. 1222, pp. 213–228.

Duggan, D. 1997. A type-based implementation of a language with distributed scope. In *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, Lecture Notes in Computer Science, no. 1222, pp. 277–294.

Engler, D. R. 1996. VCODE: A retargetable, extensible, very fast dynamic code generation system. *PLDI'96*, pp. 160–170.

Franz, M. 1997. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, Lecture Notes in Computer Science, no. 1222, pp. 263–276.

Farmer, W. M., Guttman, J. D., and Swarup, V. 1996. Security for mobile agents: Authentication and state appraisal. *Proceedings of the Fourth European Symposium on Research in Computer Security*. Rome, Italy, Springer-Verlag Lecture Notes in Computer Science, no. 1146, pp. 118–130.

Farrow, R., Marlowe, T. J., and Yellin, D. M. 1992. Composable attribute grammars: Support for modularity in translator design. *ACM 1992 Principles of Programming Languages*, pp. 223–234.

Gosling, J. 1995. Java intermediate bytecodes. *ACM Workshop on Intermeidate Representation*, pp. 111–118.

Gray, R. S. 1996. Agent Tcl: A flexible and secure mobile agent system. *Proceedings of the Fourth Annual Tcl/Tk Workshop*. Monterey, CA, pp. 9–23.

Gibson, P. A., and Stoyenko, A. D. 1992. Development and integration of a concurrently executing interactive user interface for the I-STAT portable clinical analyzer: A case study in real-time systems integration. *J. Sys. Integration* 2(4): 349–388.

Glagowski, T. G., and Jones, K. L. 1995. A new method for implementing fuzzy retrieval from a spatial database. School of Electrical Engineering and Computer Science, Washington State U., Pullman, WA, submitted to *Information Sciences*.

Ghezzi, C., and Vigna, G. 1997. Mobile code paradigms and technologies: A case study. *Proceedings of the First International Workshop on Mobile Agents.* Berlin, Germany.

Halang, W. A., and Stoyenko, A. D. 1991. *Constructing Predictable Real-Time Systems.* Kluwer Academic Publishers, with a preface by Konrad Zuse.

Harelick, M. S., Marlowe, T. J., Stoyenko, A. D., and Sinha, P. 1995. A constraint function classification for complex systems development. *1995 Complex Systems Engineering and Assessment Technology Workshop*. Ft. Lauderdale, FL.

Hinchey, M. G., and Bowen, J. P. (Eds.) 1995. *Applications of Formal Methods*. Prentice Hall International Series in Computer Science, Hemel Hempstead, with a foreword by C.A.R. Hoare.

Hoang, N. D. 1991. The essential views of systems development. *Proceedings of 1991 Systems Design Synthesis Technology Workshop.* Naval Surface Warfare Center, Silver Spring, Maryland, pp. 3–9.

Hoover, J. 1992. Alphonse: Incremental computation as a programming abstraction. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

Houstis, C. E. 1990. Module allocation of real-time applications to distributed systems. *IEEE Transactions on Software Engineering* 16(7): 699–709.

Stoyen, A. D., Marlowe, T. J., and Petrov, P. 1996. Heterogeneous debugger-monitor-corrector, working report.

Jones, N. D., Gomard, C. K., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

Kligerman, E., and Stoyenko, A. D. 1986. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering* 12(9): 941–949.

Laplante, P. A., Marlowe, T. J., and Stoyenko, A. D. 1996. Language mechanisms for real-time image processing. *Control Engineering Practice*.

Levi, S.-T., Mosse, D., and Agrawala, A. K. 1988. Allocation of real-time computations under fault tolerance constraints. *Proceedings of the IEEE 1988 Real-Time Systems Symposium*, pp. 161–170.

Lewis, B. T., Deutsch, L. P., and Goldstein, T. G. 1995. Clarity Mcode: A retargetable intermediate representation for compilation. *ACM Intermediate Representation Workshop*, pp. 119–128.

Liu, Y. A., and Teitelbaum, T. 1995. Caching intermediate results for program improvement. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 190–201.

Lo, V. M. 1988. Algorithms for task assignment in distributed systems. *IEEE Trans. Computers* 37(11): 1385–1397.

Marlowe, T. J., and Ryder, B. G. 1990. An efficient hybrid algorithm for incremental data flow analysis. *17th Annual ACM Symposium on the Principles of Programming Languages*, pp. 184–196.

Marlowe, T. J., Stoyenko, A. D., Masticola, S. P., and Welch, L. R. 1994. Schedulability-analyzable exception handling for fault-tolerant real-time languages. *Real-Time Systems* 7(2): 183–212.

Marlowe, T. J., Stoyenko, A. D., Laplante, P., Daita, R. S., Amaro, C. C., Nguyen, C. M., and Howell, S. L. 1996. Multi-goal objective functions for optimization of task assignment. *Control Engineering Practice*.

MPI-2: Extensions to the message-passing interface. 1996. *MPI Forum*.

Marlowe, T., and Ryder, B. 1991. Properties of data flow frameworks: A unified model. *Acta Informatica* 28(2): 121–164.

Marlowe, T. J., Stoyenko, A. D., Welch, L. R., Laplante, P., and Masticola, S. P. 1993. Incremental analysis for reuse and change in a software development environment for hard-real-time systems. *IEEE RTOS.*

Masticola, S. P., Marlowe, T. J., and Ryder, B. G. 1995. Multisource data flow problems. *ACM Transactions on Programming Languages and Systems* (5): 777–803.

Prastowo, B. 1995. Derivation of incremental Datalog programs, Ph.D. Thesis, Queens University, Department of Computer Science, Kingston, Ontario.

Real-Time Computing Lab at NJIT, 1993–1996. A hierarchy and general equational form for cost and objective functions for complex real-time systems, an ongoing report.

Reps, T. 1988. *Generating Language-Based Environments.* ACM Distinguished Dissertation, MIT Press.

Shenoi, S., Melton, A., and Fan, L. T. 1992. Functional dependencies and normal forms in the fuzzy relational database model. *Information Sciences* 60: 1–28.

Silberman, A., and Marlowe, T. J. 1996. A task graph model for design and implementation of real-time systems, to appear in *Second IEEE International Conference on Engineering of Complex Computer Systems.* Montreal, Canada.

Sreedhar, V. C., Gao, G. R., and Lee, Y.-F. 1995. A new approach to exhaustive and incremental data flow analysis using DJ graphs/ McGill University Technical Report ACAPS Memo 95.

Stone, H. S. 1977. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering* SE-3(1): 85–93.

Stoyen, A. D., Petrov, P., Marlowe, T. J., and Chiara, J. J. 1996 and 1997. A system synthesis tool for complex, embedded real-time systems, progress reports, DARPA SBIR 962-059, 21st Century Systems, Inc.

Stoyenko, A. D. 1987. A real-time language with a schedulability analyzer. Ph.D. Dissertation, Department of Computer Science, University of Toronto.

Stoyenko, A. D., Marlowe, T. J., and Laplante, P. A. 1996. A description language for engineering of complex real-time systems. *J. Real-Time Systems*.

Stoyenko, A. D., Marlowe, T. J., and Younis, M. F. 1995. A language for complex real-time systems. *Computer Journal* 38(5).

Stoyenko, A. D., Laplante, P. A., Harrison, R., and Marlowe, T. J. 1994. Engineering of complex systems: A case for dual use and defense technology conversion. *IEEE Spectrum* 31(11): 32–39.

Stoyenko, A. D., and Baker, T. 1994. Real-time schedulability-analyzable mechanisms in Ada9X. *Proceedings of the IEEE*.

Stoyenko, A. D., Welch, L. R., Laplante, P. A., Marlowe, T. J., Amaro, C., Cheng, B.-C., Ganesh, A. K., Harelick, M., Jin, X., Younis, M., and Yu, G. 1993. A platform for complex real-time applications. *1993 Complex Systems Engineering and Assessment Technology Workshop*. Beltsville, Maryland, USA.

Stoyenko, A. D. 1991. General model and mechanisms for heterogeneous model-level RPC interoperability. *IEEE 1991 Symposium on Parallel and Distributed Processing*. Dallas, Texas, USA, pp. 668–675.

Stoyenko, A. D. 1994. SUPRA-RPC: SUBprogram PaRAmeters in remote procedure calls. *Software—Practice and Experience* 24(1): 27–49, earlier version in *IEEE SPDP'90*.

Stoyenko, A. D., and Halang, W. A. 1993. High-integrity PEARL: A language for industrial real-time applications. *IEEE Software* 10(4): 65–74.

Stoyenko, A. D., Marlowe, T. J., Halang, W. A., and Younis, M. 1993. Enabling efficient schedulability analysis through conditional linking and program transformations. *Control Engineering Practice* 1(1): 85–105.

Stoyenko, A. D., Marlowe, T. J., Cheng, B.-C., and Ganesh, A. Performance prediction functions for real-time software components. In consideration for *IEEE Transactions on Parallel and Distributed Systems*.

Stoyenko, A. D., and Marlowe, T. J. 1992. Polynomial-time transformations and schedulability analysis of parallel real-time programs with restricted resource contention. *J. Real-Time Systems* 4(4): 307–329.

Stoyenko, A. D., Hamacher, V. C., and Holt, R. C. 1991. Analyzing hard-real-time programs for guaranteed schedulability. *IEEE Transactions on Software Engineering* 17(8): 737–750. Earlier version in *IEEE RTSS'87*.

Stoyenko, A. D., Marlowe, T. J., Younis, M. F., and Petrov, P. 1997. A language support environment for complex distributed real-time applications. In *Proceedings of the Third IEEE International Conference on Engineering of Complex Computer Systems*, extended version invited and submitted to *IEEE Transactions on Software Engineering*, IEEE Computer Society Press.

Stoyen, A. D., and Laplante, P. A. (Eds.) 1995. *Engineering of Complex Computer Systems: Fundamentals, Techniques & Applications. IEEE Press*.

Stoyenko, A. D., Bosch, J., Aksit, M., and Marlowe, T. J. 1996. Load balanced mapping of distributed objects to minimize network communication. *J. Parallel and Distributed Processing* 34(2): 117–137.

Stoyenko, A. D. 1992. Evolution and state-of-the-art of real-time languages. *J. Systems and Software* 18: 61–84.

Stoyenko, A. D., and Georgiadis, L. 1992. On optimal lateness and tardiness scheduling in real-time systems. *Computing* 47: 215–234.

Strom, R., and Yemini, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. on Software Engineering* 12: 157–171.

Trivedi, K. S., Sahner, R., and Puliafito, A. 1995. *Performance and Reliability Analysis of Computer Systems An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers.

Wei, Y.-H., Stoyenko, A. D., and Goldszmidt, G. S. 1991. The design of a stub generator for heterogeneous RPC systems. *J. Parallel and Distributed Computing* 11: 188–197.

Yellin, D. M. 1993. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica* 30: 369–384.

Yellin, D. M., and Strom, R. E. 1991. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems* 13(2): 211–236.

Younis, M. F., Marlowe, T. J., and Stoyenko, A. D. 1994. Compiler transformations for speculative execution in a real-time system. *IEEE 1994 Real-Time Systems Symposium*. San Juan, Puerto Rico.
Younis, M. F., Tsai, G., Marlowe, T. J., and Stoyenko, A. D. 1995. Formal verification for speculative execution in real-time systems. *First IEEE International Conference on Engineering of Complex Computer Systems.*

Alexander D. Stoyen is Endowed Full Professor of Computer Science and Director, Center for Management of Information Technology at the University of Nebraska at Omaha. He received his Ph.D. in Computer Science from the University of Toronto in 1987. Upon graduation, Dr. Stoyen joined IBM T. J. Watson Research Center as a research staff member. In 1990, he became an Assistant Professor of Computer Science at the New Jersey Institute of Technology, where he built the Dependable Real-Time Systems Laboratory. At the time of leaving NJIT in January 1999, Dr. Stoyen was an Associate Professor of Computer Science and of Electrical and Computer Engineering. Dr. Stoyen has held visiting, consulting and advisory appointments at universities, corporations and other organizations throughout North America, Europe and Asia. He has contributed significantly to a number of key technological concepts in real-time systems and has published more than 110 articles in journals, books and conference proceedings. Dr. Stoyen has served as an organizer of many professional events, chair of IEEE Technical Committees, and editor or guest editor of book series and archived peer-reviewed journals, including the *IEEE Computer, J. Real-Time Imaging* and *J. Real-Time Systems*. He is also founding CEO and CTO of 21st Century Systems, Inc., a company researching, prototyping and developing revolutionary tools for construction and operation of complex, large-scale software applications. Dr. Stoyen is a Senior Member of the IEEE and is an IEEE Computer Society Golden Core Member.

Dr. Petrov received his Ph.D. in Computer Science from New Jersey Institute of Technology in 2000 for his research on Agent-based Real-time Decision Support Systems. While at NJIT, Dr. Petrov was part of the Dependable Real-time Systems Laboratory, where he contributed to multiple ONR and NSF sponsored research projects on Real-time languages, compilers, tools and systems engineering. Dr. Petrov graduated (summa cum laude) from University of Central Florida with a Bachelor of Science degree in Computer Science. During his last year in UCF Dr. Petrov participated in a STRICOM-sponsored Computer Generated Forces program at the Institute for Simulation and Training, Orlando, FL. Currently Dr. Petrov is a Vice President, Technology & Development at 21st Century Systems, Inc., a growing small business focusing on innovative software technology complex military and other command and control problems. Plamen Petrov has lead a number of successful R&D and projects at 21CSI while maintaining his research interests in compile-time analysis, complex distributed systems, mobile code, decision support systems, and computer-based command and control.