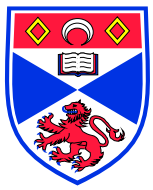


A Meta-Programming Framework for Software Evolution

by

Katherine Elizabeth Mickan Enderling

Honours Computer Science (First Class),
The University of Adelaide, Australia, 2000

B.Sc. (Mathematical & Computer Sciences),
The University of Adelaide, Australia, 2000

Thesis submitted for the degree of

Doctor of Philosophy

in

The School of Computer Science
The University of St Andrews

November, 2005

Contents

Abstract

Software systems are expensive to build and deploy, but their effectiveness diminishes over time unless they are able to meet changing user requirements. This work is motivated by the need to build, understand and manage software systems that can evolve in order to adapt to changing environmental demands. Evolvable software requires change management in the form of processes that determine when and what to change and mechanisms to effect those changes.

This thesis introduces the Meta-Programming Framework (MPF), which provides mechanisms for automatic evolution, allowing meta-programs, or management components, to introspect and evolve existing systems. A unique combination of technologies supports an evolution process whereby a meta-program stops the relevant part of an executing program using a decomposition operator, obtains and evolves a representation of it, and incorporates the changes back into the executing system using structural reflection. During the evolution process, introspection enables a meta-program to obtain a Hypercode graph of any value in the system. Hypercode graphs present a complete, up-to-date representation by encompassing both program syntax and closure. The implemented framework defines an interface for meta-programs to traverse, manipulate and evolve Hypercode graphs. The evolution may proceed without the evolved values losing their internal state, because it is preserved by the Hypercode graph representation.

The key contributions of this approach are the support for incremental evolution, and the Hypercode graph representation that provides introspection as well as giving meta-programs the flexibility to both update existing values and introduce new behaviours. In addition, because the MPF is constructed in itself, it can be evolved in the same way as any other program. As part of an evolution framework that includes change management tools, the MPF facilitates the automatic evolution of software systems.

Statement of Originality

I, Katherine Mickan, hereby certify that this thesis, which is approximately 44 000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

Date | Signature of Candidate

I was admitted as a research student in January 2002 and as a candidate for the degree of Doctor of Philosophy in January 2002; the higher study for which this is a record was carried out in the University of St Andrews between 2002 and 2005.

Date | Signature of Candidate

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date | Signature of Supervisor

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker.

Date | Signature of Candidate

Acknowledgments

I would like to sincerely thank my supervisors Prof. Ron Morrison and Dr. Graham Kirby for their input, guidance and insightful comments. It has been excellent to have the opportunity to work with them.

I would like to thank Dr. Dharini Balasubramaniam for her supervision and useful discussions.

I would like to thank the other PhD students and people in the department for their friendship and help.

I would like to thank Dr. Dave Munro from the University of Adelaide for setting me on this path in the first place.

For financial support I would like to thank the EC Framework V project ArchWare (IST-2001-32360) and the ORS Award Scheme.

For his generosity and bad jokes I would like to thank Dr. Frank Gunn-Moore - without him I would never have got so far.

I would like to thank Norman and Alison Lillie for their kindness and for giving me a home.

I would like to acknowledge the invaluable support of friends, especially Emily and Mark.

Most of all I would like to thank Stefan for being understanding and supportive, providing me with encouragement, and making me happy, and my family who have always looked after me even though I am a long way away.

Katherine Mickan. November 2005.

Chapter 1

Software Evolution

This chapter discusses software evolution, why it is a relevant problem for software systems, how it is defined, and different approaches to solving it. These include self-adaptive software, meta-programming and software architectures. Finally, a set of characteristics of frameworks that aim to support software evolution is defined. Existing research will be compared against these in Chapter 2.

Complex and dynamic software systems either evolve or their efficacy deteriorates until they fall out of use. The term *co-evolution* is used to describe the co-dependent relationship of dynamic change in commercial environments and the software that supports them (Greenwood et al., 2000). As requirements change, software needs the capacity to adapt to the altered environment in which it is used in order to avoid increasing redundancy. Businesses, in turn, need to adopt new opportunities offered by advances in software, for example, by opening up sales to an expanded market through selling merchandise on a website.

Making use of the evolutionary potential in software is particularly important for large, long-lived systems which are expensive to build and deploy. Lehman (1996) defines eight laws representing a theory of the process of software development and evolution. The first law is the law of continuing change, stating that a system must be continually adapted or it will become progressively less satisfactory. The second law states that as a system is evolved, its complexity will increase unless work is done to maintain or reduce it. These two laws indicate why we need to develop systems that support software evolution. Co-evolution, by taking the business process into account, can assist in determining which parts of the software are no longer needed and can be removed to achieve a less complex system.

The problem domain the work in this thesis aims to address is the development of software that can adapt to a changing environment. Such software is able to incorporate new functionality and thereby remain useful in a changing business and user environment. It is also able to recover from unexpected failure - complex software systems can exhibit emergent behaviour, which, by definition cannot be predicted at development time. This can occur once the software has already been deployed and requires live fixing. Ideally, the part of the software that has failed can be isolated and fixed without interfering with the rest of the system.

This thesis focuses on systems which have the following properties:

- They are long lived and it is therefore inevitable that requirements will change over time and the system will need to evolve to meet the new requirements.
- They execute continuously and system down-time is not an option. Examples such as the telephone network or a travel booking system are in constant use. In addition to requiring continuous execution, these systems have internal data,

which should be preserved over change. For example, the current state of a holiday that has been half booked should not be lost when the booking component is evolved.

- These systems are component based, allowing evolution to take place in one part of the system without interfering with the execution of the rest of the system, i.e., change can be isolated.

Some examples of systems in the problem domain are continuously running business process models, GRID applications, peer-to-peer routing systems and control systems.

Currently, research focuses on adapting systems to deal with situations such as variable resources, e.g., changing bandwidth and availability of services, system errors, e.g., component and connection failure, and changing user priorities, e.g., requirements for either high or low bandwidth streaming. The work is limited in that it deals with change that can be predicted at development time and the mechanisms to perform adjustments can be included then.

This thesis, on the other hand, is concerned with the evolution of systems. Evolution includes the ability to respond to emergent properties exhibited by complex systems. Because these cannot be predicted, the change mechanisms have to be flexible and include developers who can interpret and develop solutions for emergent problems. Additionally, evolving systems should be able to incorporate new functionality in response to changing requirements. These two kind of changes should be unconstrained. That is, change should be able to be applied equally to any part of the system whether it is a large component or a single function. It should not be restricted to a particular type of change, for example, component addition, replacement or variable updates.

1.1 Control Systems

The challenge faced by software systems, which must evolve to stay useful in a changing application environment, is also faced in hardware. Engineers use *control systems* to model adaptation. Feedforward control systems are used to adjust a process to changing environmental input before its output is affected (Brosilow and Joseph, 2002).

In a control system, the feedforward controller is the component which adjusts the process in response to environmental inputs. The feedback controller makes adjustments

that are dependent on feedback from the process itself. Feedforward control is applied when there is a disturbance that can be measured and eliminated before it affects process output. It is always used in conjunction with feedback control, which examines process output and alters the process to ensure its continuing correctness. A feedback and feedforward control system is illustrated in Figure 1.1, which shows that the feedforward controller responds to environmental input and the feedback controller reacts to the process output.

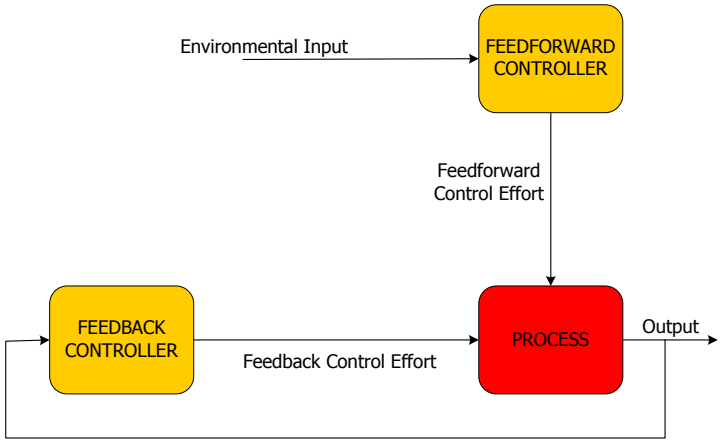


Figure 1.1. Control System. A feedforward and feedback control system.

Applying the control systems paradigm to software (Lehman, 1996), a software system must adjust to user and business needs that diverge from the original requirements over time. In order that the software remains useful, the adjustments should be performed before the software becomes ineffective to users. Therefore, *"the software evolution process . . . constitutes a complex feedback system"* (Lehman, 1996). In the sphere of software evolution the controllers are *adaptive engines* and the process is a software system. An adaptive engine is comprised of the components that drive evolution.

Open software systems can incorporate new functionality into their evolution (Oreizy et al., 1999) to deal with alterations in the environmental conditions, such as changing user requirements. Feedforward controllers correspond to adaptive engines in open systems. Lehman's *positive feedback*, which is defined to trigger or accelerate growth, would be input to an open adaptive engine. The evolution of open systems is only limited by the policies and mechanisms in the adaptive engine. These constrain the

input it can receive from the environment and how it can interpret and act on that input.

Closed software systems, like feedback control systems, detect changing system outputs or environmental inputs, such as Lehman's *negative feedback*, and react by applying modifications to stabilize the system. As with feedforward and feedback control systems, software applications use both open and closed adaptive engines in the same system. The adaptive engine in a closed system makes modifications to maintain a functioning system or level of performance, endeavouring to ensure that the system operates within a set of constraints. It cannot introduce new behaviours and is therefore unable to respond to unanticipated business and user needs. Closed systems are also incapable of addressing the effects of emergent properties in complex systems.

The input to both open and closed adaptive engines may come from within the software as well from its environment. The difference is that an open adaptive engine can access new functionality and incorporate it into the evolving system, whereas a closed adaptive engine adjusts existing functionality to regulate or improve its performance.

Emergent phenomena are unexpected and unpredictable from a description and may appear irrespective of changes in the external environment. Because a closed system cannot incorporate new behaviour, it is unable to deal with these properties which were unforeseeable at the time the adaptive engine was developed. Open systems can incorporate new behaviour to deal with emergent properties.

1.1.1 System Monitoring

When the control system model is applied to software evolution, the controllers in Fig. 1.1 become adaptive engines, which make changes motivated by system outputs and environmental inputs. The gathering and interpretation of system outputs requires some form of monitoring. This is commonly implemented using probes and gauges.

Probes monitor an executing system to collect information such as the load on a server or the bandwidth of a network path. Ideally probes should be non-intrusive and designed for easy monitoring (Garlan and Schmerl, 2002). Gauges receive measurements from the probes and interpret them according to a system model. If the measured values fall outside an acceptable range, an alarm is raised to trigger mechanisms for self-adaptation.

One of the benefits of using probes and gauges is that they can externalize system monitoring. A number of tools have been developed, which implement probes by collecting information about a system and disseminating it to external tools or components. Some of these monitor existing applications unobtrusively, for example, Remos (DeWitt et al., 1998), which can be run on a separate machine from the application it is measuring, has been used in Garlan et al. (2001) to gather information about host and network loads. Another approach is to insert probes into an existing application. For example, in Discotect (Schmerl et al., 2005), probes to monitor object instantiation have been inserted using AspectJ (Kiczales et al., 2001), which allows code fragments to be weaved into Java byte-code. Probes can also be inserted at development time, as is the case with components built using Active Interfaces (Heineman, 1997). These components have an adaptation interface along with their own interface. Probes can be associated with *before* and *after* phases of a component's methods.

Gauges interpret low-level events received from probes and generate high-level events, which are meaningful in the context of a system model such as a software architecture. Events created by a gauge can be received by a number of different management components. Conversely, a management component may require events from more than one gauge. Gauges usually depend on information from multiple probes and possibly other gauges. Their applications can be extended beyond the standard task of ascertaining whether a system conforms to a set of constraints in policy driven systems. Gauges have been developed that interpret system level events as architectural events, from which the software architecture of a running application can be discovered (Yan et al., 2004; Schmerl et al., 2005).

In an evolving system, probes and gauges should be able to change along with the system. Figure 1.2 shows one method for implementing flexible probes (Balasubramaniam et al., 2004b; Balasubramaniam et al., 2005). The diagram shows the system structured as a *sink* corresponding to the closed adaptive engine (feedback controller) and a *source* corresponding to the evolving software system (process). The source advertises an interface exposing *observations* on its state that can be made by probes. The sink uses information from the interface in constructing a probe. This probe is connected to a gauge in the sink, which will trigger some change if the observed values violate the system's constraints. The sink sends the probe to the source using a *probe connection*.

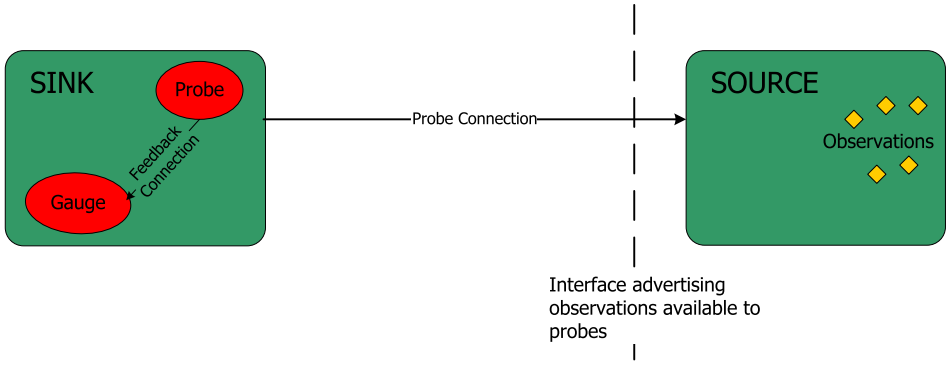


Figure 1.2. Probes and Gauges. A structure for flexible probes in an evolving system, shown before the probe is installed in the source.

On receiving the probe, the source installs it (Fig. 1.3). The probe, now inside the source, sends its feedback directly to the gauge in the sink. This flexible approach allows the sink (adaptive engine) to install probes as necessary without the source (software system) having any knowledge of the probe's internal functionality or the system's constraints.

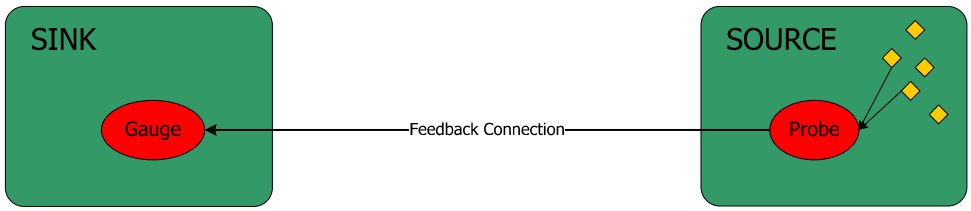


Figure 1.3. Probes and Gauges. A structure for flexible probes in an evolving system, shown after installing the probe in the source.

1.1.2 Open Adaptive Engines

An open adaptive engine in an evolving software system effects adaptations in response to environmental change. The implementation of open adaptive engines is largely an unsolved problem. Change in the business or user context is the environmental input driving the evolution and is difficult to interpret. System monitoring may also produce emergent behaviours. Consequently, it is complex to determine what software changes should result from the environmental input and system output, requiring user or developer intervention. Work in open adaptive engines predominantly concentrates on how to incorporate the changes into the system. Accordingly,

although adaptive engines are being developed that can alter the behaviour of a system to deal with emergent properties, discovering the properties and then quantifying them is complicated.

The ideal adaptive engine fulfills the following evolutionary goals:

- Changes are effected with minimal disruption to the unaffected parts of the system so they may continue to execute during the change.
- The adaptive engine has a complete, up-to-date system representation.
- Unrestricted changes may be applied to the representation and then incorporated into the process, or executing system.
- The changes can assimilate the system's existing state.

These goals are applicable to both open and closed adaptive engines, except for the third goal, which relates to unrestricted changes and is aimed at open systems. This thesis examines the provision of mechanisms to support construction of an ideal adaptation engine for an open system.

1.2 Software Architecture Based Approach to Evolution

Evolution can occur at different levels of abstraction in a software system, from assignment into mutable locations at the language level, to the changing topology of a software architecture. However, reasoning about evolution at a low level, which does not separate application functionality from the process of change, makes it difficult to manage the evolution of a complete system. A higher level of abstraction may assist reasoning about evolution, both what to change and how and when to change it. Evolution at the level of software architectures allows the complex details to be filtered out, supporting evolution of a total system in contrast to small scale changes. Design knowledge embodied in the software architecture encourages correctness in evolution (Schmerl and Garlan, 2002). Modelling a system's evolution using the software architecture requires a flexible architecture that can express and accommodate change.

1.2.1 Defining Software Architectures

Software architectures model systems at a high level of abstraction, commonly in terms of *components* and *connectors* (Oreizy et al., 1999). Details available at the source code level are hidden in favour of representing the bigger picture. Components implement application functionality and are not privy to information about who they are communicating with, or how the communication takes place. Connectors encapsulate the communication and co-ordination between components. Software architectures also have a set of *properties* defining and constraining their characteristics. Separation of communication and computation, as well as loose coupling between components encourages flexible evolution. For example, a pipe and filter architecture (Fig. 1.4) has:

- Filter components containing the application functionality.
- Pipe connectors which control the communications between filters.
- A set of properties that constrain filters to be independent of each other.

Using this architecture, a filter component can be replaced by disconnecting the pipes attached to it and reconnecting them to a new filter. The architectural properties ensure that the other components are not affected by the update.

A definition of software architecture (Garlan and Perry, 1995) has been adapted into the IEEE 1471-2000 standard (IEE, 2000) which states that a software architecture is:

> *The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.*

Software architectures are usually defined using Architecture Description Languages (ADLs). Some of these and the evolutionary mechanisms they support are discussed in Chapter 2.

Software architecture families can be defined in terms of styles, which are described as (Oreizy and Taylor, 1998):

> *Idiomatic patterns of system organization that characterize a particular application domain.*

The elements of a style are defined to be (Garlan and Shaw, 1996):

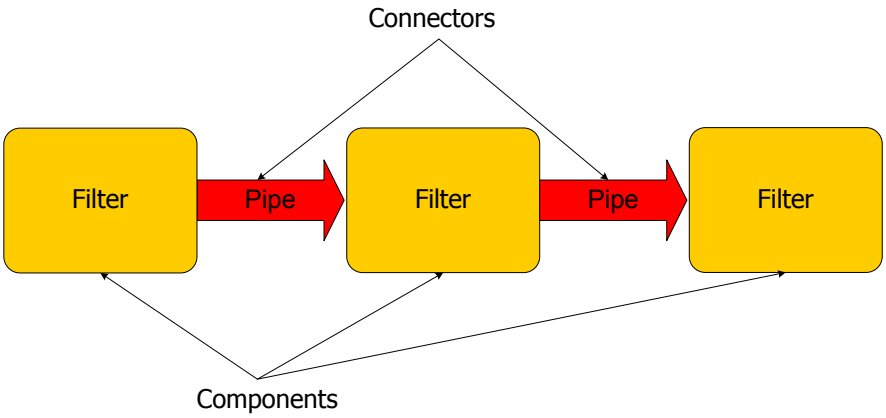


Figure 1.4. Pipe and Filter. A pipe and filter architecture is a set of components, connectors and properties.

- A *vocabulary* of component and connector types.
- A set of *constraints* on how the components and connector types may be combined.
- A set of *semantic models* which specify how to determine a system's overall properties from the properties of its parts.

Some commonly used styles include client/server, pipe and filter, object oriented and hierarchical layers. Styles may also be used to define process control systems, e.g., feedback control loops (Garlan and Shaw, 1996). A software system may contain a number of styles, which may apply to separate parts of the system or be applied hierarchically. For example, a large component may contain an internal style, as well as being part of a different style in a wider context.

1.3 Self-Adaptive Software

Self-adaptive software adjusts itself in response to stimuli from its environment without programmer interaction. Self-adaptation at a low level is inherent in programming

languages and methods, for example, exception handling and conditional expressions. Other commonly used forms of self-adaptation are:

- Online algorithms, e.g., paging algorithms chosen depending on current usage.
- Generic and parameterized algorithms.
- Machine learning techniques, which generate algorithms dependent on input data (Oreizy et al., 1999).

These forms of adaptation present a limited range of possible changes. Adaptation at a global level, for instance, the re-arrangement of a client/server system when a server goes down, or detection of degrading performance, can be provided by approaching the adaptation from a software architecture perspective. The wider perspective also offers opportunities for change management (Sec. 1.5.4), allowing the above mentioned forms of self-adaptation to be incorporated into the system.

A system may undertake self-adaptation in order to (Oreizy et al., 1999):

- Improve performance.
- Recover from failure.
- Accommodate new functionality.

Closed systems will only adapt under the first two conditions. The third condition requires an open system that can incorporate information from its environment.

The information triggering an adaptation and the interpretation of that information are crucial to a self-adaptive system, determining how and when the system will change. Therefore, a key issue in designing self-adaptive systems is deciding what information should be gathered about the system and how that information should be modelled. It defines the ontology for the view of the system available to adaptive elements.

Interpreting the information with respect to the software architecture is complex. Constraints and properties of the software architectural model should be chosen carefully as they limit the set of problems which can be detected. When adaptation becomes necessary or possible, a repair strategy needs to be chosen which will improve the system, or at least keep its performance constant. However, while constraints are useful in determining if a system is operating within a set of parameters, it can be difficult

to determine whether a complex system has improved or not. A system that allows for evolving probes, constraints, architectural properties and repair strategies can approach evolution as a learning process adapting along with the system.

1.3.1 Autonomics

Autonomics is a sub-field of self-adaptive software, which aims to create software systems that can adjust themselves without programmer intervention. The name is drawn from the autonomic nervous system which regulates involuntary functions in the human body. Autonomic systems aspire to be self-configuring, self-healing, self-optimizing and self-protecting (Kephart and Chess, 2003; Kephart, 2005).

Conventionally, autonomic systems are made up of a set of *autonomic elements* (Kephart and Chess, 2003). An *autonomic element* is comprised of a *managed element* under the control of an *autonomic manager*, as shown in Fig. 1.5. The *managed element* contains the application functionality. The *autonomic manager* is responsible for managing the state, behaviour and interactions of the *autonomic elements*. *Autonomic elements* interact with their environment via signals and messages and are part of a network of agents. In an evolving system the *autonomic manager* could be evolved by other agents, or evolve itself, possibly in response to stimuli from other agents.

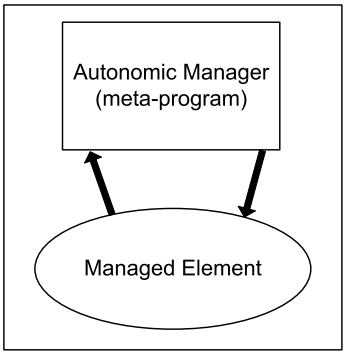


Figure 1.5. Autonomic Element. An autonomic element contains a manager and a managed element.

Autonomic systems attempt to support both fine-grained evolutions and re-organizations motivated by high level policies retained in each element. In the Unity system developed at IBM Research (Chess et al., 2004), each element should be as far as possible self-managing, able to heal internal failures, able to optimize its own behaviour,

and able to protect itself from attack. Individual elements do not maintain an overview of the entire system. Instead there is a central registry to which they can refer to find services offered by other elements. To exchange services, an element establishes a relationship with other elements it has located using the registry. Another example of an autonomic system where elements are as independent as possible is Self-Tuning Web Servers, such as (Diao et al., 2003). The servers adapt to stay within a set of performance parameters. Adaptation is performed without global knowledge of the system in which they operate.

1.4 Meta-Programming

Meta-Programs are defined to be *programs which manipulate other programs or themselves* (Czarnecki and Eisenecker, 2000). In general, meta-programs can be either *imperative* or *declarative* (Czarnecki and Eisenecker, 2000). Declarative programming focuses on *what* a program should do. Imperative programming, on the other hand, defines *how* a program should achieve its goal. Declarative meta-programs are easier to check formally and the programmer does not have to reason about low-level details while writing them. However, mapping a declarative program to compiled code is a complex task. Imperative meta-programs may be difficult to write and require detailed knowledge about the underlying system, but they also allow the programmer more control over how the high level policies are realized. Commonly, software systems are designed so that the higher layers are defined declaratively (in the software architecture) and the lower layers imperatively (in the implementation).

Meta-programming environments are frameworks or languages where meta-programs can be constructed and executed. A framework is defined here to be a system, or set of components, that operates over another system. Environments that offer support for generative programming allow changes to be made to existing code in a partially automated way. Knowledge about evolution, such as when and how it should take place, can be captured by the environment.

Another system, which takes advantage of Meta-Programming to evolve an executing system, is a framework for refactoring (Ebraert et al., 2004). The framework uses Smalltalk (Goldberg and Robson, 1983) and is designed with a two level architecture. The top level (meta-program) gathers information about the activities of the lower level components by monitoring their communication. On deciding to make a change,

Smalltalk's reflective facilities are used to add, remove and change methods, add and remove instance variables, and change class names. To preserve consistency, the application must be forced into a quiescent state for the meta-program to perform the evolution. This involves stopping the entire application and highlights the need for a *decompose* operator (Sec. 5.3.1) to evolve systems where the execution should be continuous.

1.5 Characteristics of Evolving Systems

Three stages in the evolution of software architectures have been defined (Morrison et al., 2004). Firstly, deciding *when* change is required, that is, appropriately responding to some stimulus. Secondly, deciding *what* change is required, that is, determining the appropriate change with respect to the stimulus and the environment. Thirdly, deciding *how* change is applied and using the appropriate mechanisms to apply it.

This section discusses the *when*, *what* and *how* of evolution. It also considers *where* evolution takes place, *who* initiates evolution, and management of the change process.

1.5.1 When to Evolve

Dynamically or Statically

Software can be evolved either during execution (dynamic), or by stopping the entire system and replacing it with a new system incorporating the required alterations (static). An example of an evolution which stops the execution is an operating system update requiring the user to restart their computer in order to complete the installation. This approach requires some down time and is therefore not suitable for applications where the system must be available at all times, such as air traffic management. Another drawback to static change is that internal system state is lost when the execution is stopped. Solutions to this problem usually require internal data to be written out to a file or database, which is then used to restore the system state on restart.

Dynamic evolution is appropriate for systems that are large, complex and long lived. In order to support dynamically changing architectures (Medvidovic, 1996):

1. Changes can be specified after a system has been built.

2. Parts of an architecture can be modified without having to generate the whole system again.

3. An architect can view parts of an architecture without having to see the whole system.

Commonly, work at the software architecture level avoids addressing the second point by assuming that the mechanisms for reflecting dynamic changes into the system will be provided by the implementation. Systems implemented using reflection are exceptions, for example, ArchJava (Sec. 2.4) and ArchWare (Sec. 3.7.4).

A dynamic system should be able to perform the following actions at runtime:

- Introduce both extant and newly defined components.
- Update components while maintaining their state.
- Bind and unbind connections between components.

The two basic requirements for a system supporting dynamic addition and removal of components are dynamic linking, and facilities to alter connections between components (Oreizy et al., 1998). Ideally, the above actions are defined in the ADL, and supported by the underlying system, thereby exposing the mechanisms for evolution at the higher level of abstraction.

Reflective programming languages such as Lisp (Anderson et al., 1986), Smalltalk (Goldberg and Robson, 1983) and ProcessBase (Morrison et al., 1999a) support runtime change at a fine granularity. In order to support these changes at the architectural level, ADLs can leverage the techniques used in these programming languages (Medvidovic, 1996). Reflective mechanisms increase the evolutionary power of systems at any level of abstraction. However, most ADLs concentrate on the software architecture as a separate entity from the running system. The implementation is relied upon to provide mechanisms to evolve the running system, but these are not integrated into the architecture or the change management process.

Lazy or Opportunistic

The time at which an evolution takes place can be determined either *lazily* or *opportunistically* (Oreizy et al., 1999). *Lazy* evolution takes place only when change becomes imperative in order to fix a problem. *Opportunistic* evolution is performed when possible, for example, to improve degrading but acceptable performance.

1.5.2 What to Evolve - Change Categories

One proposed set of changes that may be made to a software architecture at runtime is (Medvidovic, 1996):

1. Addition of new components.
2. Upgrade of existing components.
3. Removal of unnecessary components either temporarily or permanently.
4. Reconfiguration of software architecture by disconnecting and connecting components and connectors.
5. Reconfiguration of system architecture by modifying the mapping of components to processors.

In Morrison et al. (2004), runtime change is classified into three categories. *Dynamic* change is defined as change in the topology of the components and their interactions, including the creation of new, predefined components during execution. This definition subsumes 4, above. *Update* change is the replacement of components, as in 2, above. *Evolutionary* change is the alteration of component and interaction specifications during execution. It includes 1, above, but goes further to address changing architectural styles and constraints.

A flexible software system, which can evolve to meet the needs of users under various conditions of use, makes changes which either specialize the software making it more useful to a subset of potential users, or generalize it making it appropriate for a larger set of users. Some researchers argue that software should be able to meet the needs of all users, both specialized and general (Kiczales et al., 1993).

Preservation of State

Support for preservation of internal state over change is a desirable property of systems that support dynamic update. One way to achieve this is for components to be able to divulge and install state information (Hofmeister, 1993). This is the approach taken by C2 components (Sec. 2.6). An alternative, more flexible approach, which does not require components to conform to any particular interface is taken by Hypercode (Sec. 3.5), where state is preserved by maintaining a representation of program closure using

links to extant values in program code. Program closure is the set of a program and all the values that are used in its execution. This includes that program's data other programs that are called during its execution. For example, a function closure is the function code and a representation of the function's lexical environment (i.e., the set of available values) at the time when the function was created.

1.5.3 How to Evolve - Change Processes

There exists a spectrum of techniques for implementing evolution. These range from systems using edit, re-compilation and re-binding through autonomics and component based evolution. In the most basic evolution the execution is halted, the source code edited, and the entire program recompiled and restarted. An advancement on basic evolution allows certain values to be updated on the fly. However, the values that can be changed are determined at development time. In the first step towards incorporating new behaviours at runtime, evolution can be planned and included as part of the software life cycle, causing less disturbance during changes. The least disturbance is caused in systems that permit components to be replaced during execution. In addition, the ideal evolving system can also be structurally evolved at runtime. This involves reorganizing components' connectivity.

The types of evolution needed in an open adaptive engine are: component replacement and structural evolution. The former incorporates change types 1 - 3 in Section 1.5.2 and the latter covers 4 and 5 as well.

Basic

The basic process of constructing a system involves writing some source code and then compiling, binding and executing it (Fig. 1.6). The most basic evolution entails subsequently changing the source code (assuming it is still accessible), then recompiling, binding and executing what is effectively a new system. The evolution results in system down time and any internal program state and data are lost. A system model in the form of a software architecture, which could ensure correctness or guide understanding of the system before and after the change, is not used for these purposes.

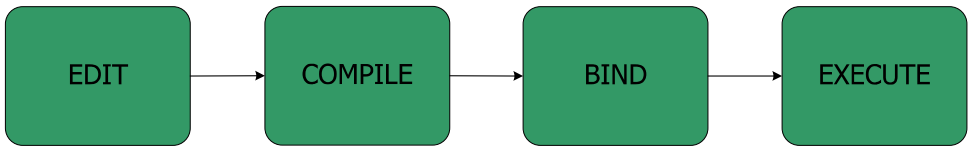


Figure 1.6. Basic evolution. The edit - compile - bind process.

Basic Update

An improvement on basic evolution is to design a system so that some of its parts can be updated on the fly. This is one approach taken by autonomic systems, where components have a set of properties which an adaptive engine, or management component, can *get* and *set*. The adaptive engine reads the values of a component's properties and uses them to determine whether changes should be made. Changes are applied by updating properties using the *set* methods. The advantage over the basic type of evolution is that changes can be made without stopping the execution. The changes are, however, restricted to a pre-defined set. Basic update is therefore useful for adapting or optimizing a system, but incapable of incorporating unpredicted changes.

Planned Evolution

Advancing on techniques for basic evolution requires a degree of planning to incorporate evolutionary mechanisms into the original system. In order to evolve a system and salvage existing program state, developers must have some way of serializing the information and writing it out to the file system or a database.

Figure 1.7 shows an evolution cycle, which starts at *edit* and continues with compilation, binding and execution. The first part of an evolution involves saving internal program state and data. Then the source code must be obtained, after which it is changed in the edit phase and then compiled. After compilation, the internal state and data that was saved must be bound back into the system, for example, by reading it in from a file.

This elementary approach to planning evolution still has limitations. The entire execution is stopped to perform any changes, the consequences of which range from annoying (e.g. an operating system update) to dangerous (e.g. an air traffic control system). Internal program state is only maintained in a custom or ad-hoc way and

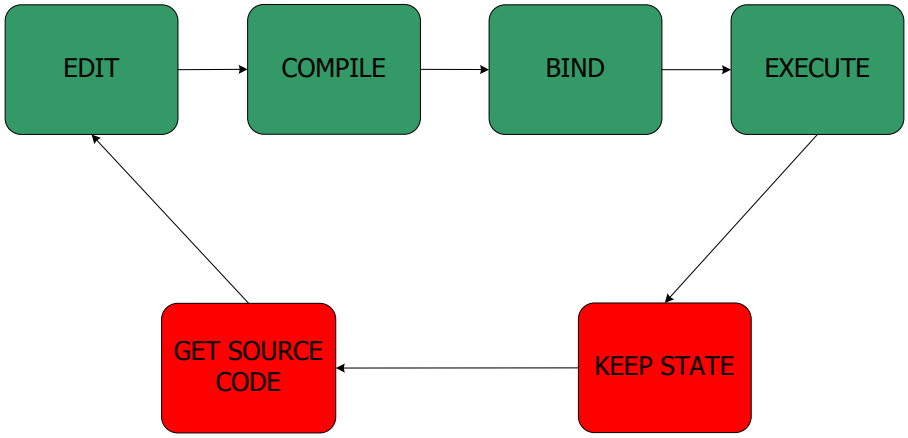


Figure 1.7. Planned evolution. The edit - compile - bind - change cycle.

might not be valid in the context of the revised source code. At development time, when the maintenance mechanisms are put in place, it could be difficult to correctly determine which data should be preserved. Finally, obtaining access to the program source code, in order to evolve it, can be difficult or impossible.

Component Replacement

Update evolution can be expanded from the basic approach, involving setting components' properties, to the replacement of the components themselves. A system capable of component replacement can incorporate new behaviours at runtime, an important step closer to an ideal setup for evolution. The realization of component replacement requires mechanisms for change localization that permit components to be replaced without system-wide disruption (Sec. 1.5.5). Mechanisms for runtime change (Sec. 1.5.1) are also necessary. The minimum support includes dynamic binding. Ideally, a replaced component's internal state can be preserved and transferred to a new component.

Structural Evolution

Structurally evolving a system reorganizes the connections between its components. Structural evolution supports system-wide changes, usually with respect to a set of inviolable constraints. In combination with the ability to dynamically introduce new behaviours, it aids unanticipated evolutions, which may not be compatible with the existing system structure. This type of evolution suits a system modelled using behaviours and their interactions (components and connectors), because otherwise connections between components are difficult to isolate and analyse. It requires that connections between behaviours are able to be created and destroyed at runtime.

1.5.4 Who Manages Evolution?

An evolving system may be changed in ways ranging from automatic to user driven. When evolution is user driven, the human interaction may vary from a user initiating the evolution at a high level, to a developer changing the application behaviour. As more of the evolution process becomes automatic, the questions of what changes to make and when to make them become more challenging. Probes, gauges and constraints are usually used to motivate and manage changes for maintenance and optimization. This approach attempts to prevent a system degrading over time, but is only applicable to closed systems and does not address the issue of how to evolve to add new functionality.

Change management (Oreizy et al., 1999; Oreizy and Taylor, 1998) supervises the process of evolution. Change management components contain the policies determining *when*, *what*, *how* and *where* to evolve as well as *who* initiates evolution. They structure runtime change and reason about new systems to determine whether or not they represent an improvement, as well as calculating of the cost of adaptation and monitoring. A general approach to change management, including support for unplanned modifications and new behaviours, distinguishes changes to system requirements from changes to the implementation that do not alter requirements (Oreizy et al., 1998). The framework suggested by the approach comprises:

- An explicit architectural model to help identify what part of a system is going to be changed.

- A technique for specifying modification descriptions in terms of the model, in order to reason about and define change.
- Constraints to preserve system integrity over change.
- A runtime model of the software architecture that is consistent with the implementation.
- Tools to implement changes.

Understanding an Evolving System

Part of managing the process of evolution is to provide a view of the software consistent with the current state of the executing system. As an evolving system changes over time by definition a static software architecture is not sufficient. In particular, evolution driven by architecture-level constraints relies on having a model of the software architecture. Views of a software architecture are usually provided by a set of tools. These either maintain a separate architectural model, which is updated alongside the implementation, or generate a view of the architecture from the implementation.

In some systems, a model of the software architecture is kept apart from the implementation and provides a separation of concerns, e.g. (Garlan et al., 2004; Oreizy and Taylor, 1998). Architectural concerns about system structure or topology are separated from implementation details, such as application behaviour. The intention is to improve the chances of evolution progressing correctly. Taking this approach necessitates the use of tools to keep the architectural model in step with the implementation as it changes.

Some systems take the approach of providing a number of architectural viewpoints and views, as defined in the IEEE Standard (IEE, 2000), characterizing various aspects of the architecture. For example, a structural viewpoint depicts the components, ports, connectors and roles in an architecture. These views are usually connected to the implementation via a tool, or not at all, in which case they must, over time, become out of step with the evolving system. The alternative approach is to extract a model of the software architecture from a running system. This model is guaranteed to be consistent with the system at a given point in time. ArchJava (Sec. 2.4) is an example of a system that generates architectural views from the implementation.

Support for Styles

Architectural styles, in the form of constraints on the software architecture, support evolution by ensuring that changes to the system do not violate original design decisions. If these decisions were also evolved, the constraints would need to change, but this has not yet been explored in research. In most systems, it is assumed that constraints are inviolable for the lifetime of the system.

Some architectural styles are more suited to runtime reconfiguration than others (Oreizy and Taylor, 1998). For example, the boundaries of change may not be clear in nested systems where functionality is not encapsulated in components. Event-based styles are amenable to change as components are not directly bound to each other and a component is unaffected by the changes in the internal structure of its neighbours. For example, a component can be stopped and replaced without holding up the execution of components to which it provides services. Characteristics of styles supporting runtime change include:

- Asynchronous message passing.
- No assumption of shared address. space or shared thread of control to avoid component and control dependencies.
- Independent architectural layers, to constrain the scope of changes.

Shared State

Change management takes responsibility for ensuring that evolutions do not cause conflicts within a system. In an incrementally evolving system, where processes access shared values, conflicts may be caused if processes are changed and their access to the shared values is no longer properly synchronized. This is particularly likely to happen in systems where access to internal state and data is preserved as a component changes.

The use of software architectures to direct evolution can restrict the granularity of change to components and connectors in order that, by design, access to shared values is not a problem. However, the evolution of complex systems with emergent properties cannot be predicted at design time. Components may be large and changes at a finer granularity may be required.

The problem can be illustrated by the example of a pipeline system operating on a shared value. Initially, the pipeline architecture controls access to the shared value and

prevents conflicts. A meta-program, which has the power to make arbitrary changes, may change the architecture and cause access problems. A change management system can, for example, ensure that evolutions that break architectural constraints do not take place.

1.5.5 Where Evolution Takes Place

Depending on the mechanisms available for evolution in a system, changing a small part can affect the entire system's execution or just a small part of it. Ideally, the effects of change can be localized and the rest of the system can continue to execute undisturbed.

Some ADLs support change localization through connectors, which can control the flow of messages, and therefore protect the system from the effects of a change. Connectors are commonly recognized as first class entities in software architectures. A primary motivation for this is to aid designers in both their reasoning about an architecture and their re-use of architectural elements. Also, first class connectors contain an architecture's interaction protocols to separate component behaviour from interaction. For example, a component may be replaced by disconnecting it and reconnecting a new component. This task is undertaken by the connectors and does not affect other components in the system.

Connectors may be defined to use different protocols, such as asynchronous or synchronous communication, or allow components written in different languages to interoperate. Moreover, first class connectors play a role in change management. They encapsulate change management policies and allow the change scope to be established (Schmerl and Garlan, 2002), as well as localizing the effects of change (Oreizy and Taylor, 1998). An example of connectors which do both these things is in C2, where connectors take control of component replacement so it happens gradually (Sec. 2.6).

1.6 Comparing Evolution Frameworks

The construction of evolvable software systems is a complex problem and has been tackled in numerous ways. There is no single approach that is agreed to work satisfactorily. In Table 1.1 a set of categories are defined, each considering a different

aspect of the problem. The categories are distilled from the discussion on characteristics of evolving systems in the previous section and they are used in the next chapter to compare different approaches. Some of the categories overlap with those defined in (Buckley et al., 2005; Mens et al., 2003), in which case the same names have been used. However, their focus is on software systems that evolve, in contrast to this thesis, which focuses on properties of frameworks that support software evolution.

Table 1.1. Categories. This table defines a set of categories for comparing evolution frameworks.

Dynamic Change

Possible Change	What kinds of changes can be made during execution?
	Openness - Can both extant and newly defined components be introduced?
	Maintain State - Can components be updated while maintaining their state?
	Connections Bound and Unbound - Can connections be bound and unbound at runtime to add and remove components?
Reflection	Are reflective facilities available to integrate changes into the executing system?

Autonomous Change

Automation	Does the system support automatic changes without user interaction?
Automated Motivations for Change	How are the questions of *what* autonomous changes to make and *when* to make them addressed?
Automated Support for Openness	What mechanisms / policies support autonomous change that introduces new behaviours?
Model Available to Management Components	Are there mechanisms to provide an up-to-date view of the system for adaptive engines?

Change Localization

Higher Order Connectors	Are there first class connectors?
Programmable Connectors	Do the connectors encapsulate the policies for change management?
Partial Decomposition	Can part of the system be changed without stopping the rest of the system?

Change Management

Change Management	Is there a change management policy?
View of Executing System	Is there a view / model of the executing system?
Partial Views	Can a partial view of the executing system be generated?
Vocabulary for Specifying Changes	How are changes specified?
Open Adaptation	Does the framework incorporate an open adaptive engine?
Closed Adaptation	Does the framework incorporate a closed adaptive engine?
Architectural Styles	Is there support for styles?

1.7 Meta-Programming Framework

The Meta-Programming Framework presented in this thesis provides the mechanisms to develop open adaptive engines for automatically evolving software systems. Meta-programs use a set of operations to manipulate program representations. The operations are designed to enable the program's evolution and can themselves be evolved since they have been developed within the framework. The representations capture program closure and meta-programs can consequently examine components' internal state and maintain it over evolutionary changes.

The primary elements of the MPF are:

- A program representation called a *Hypercode graph*, which includes both program syntax and data and accordingly captures program closure.
- A set of operations for traversing, manipulating and evolving Hypercode graphs. These operations are themselves represented as Hypercode graphs and can be evolved.

In an open system, automatic evolution implies that the system responds to environmental changes without programmer interaction. Construction of an adaptive engine for automatic evolution requires tools to manage evolution. As part of this, the MPF addresses the problem of how evolutionary changes can be applied by providing evolutionary mechanisms for a meta-program in the form of an Application Programming Interface, API, to the running system. The interface both represents the executing system to the adaptive engine and supports evolution of the representation.

The interface presented by the MPF can be programmed over and evolved in the same way as any other component in the system. This is possible because the evolutionary mechanisms operate over program representations. The interface contains a representation of itself and therefore its own mechanisms can be used to evolve it.

Change management components using the MPF can apply their own policies to determine the process of evolution. An example considered in this thesis is a process based on a system's decomposition into components, the replacing or modifying of those components, and the re-composition of the evolved system (Morrison et al., 2000a). Hypercode graphs enable the program closure of decomposed components to be captured. In other words, they provide a way of reverse engineering the executing computation so that it may be programmed over.

Each step in the evolution process is supported by operations in the MPF interface. The operations rely on underlying system support for decomposition and the incorporation of changes into the executing system, which is modelled as components and connections. Partial decomposition stops part of the system for it to be evolved while the rest continues executing.

The types of evolution supported by the MPF, in terms of those defined in Section 1.5.3, are component replacement and structural evolution. Support for these encompasses the basic and basic update types of evolution. The flavour of evolution provided by the MPF is determined by the change management policy applied on top of it. Structural evolution requires a change management framework to direct the changes. Because the framework is generic enough to be transferred to another language, its benefits are not restricted to the current evolution model. In order to be suitable for the MPF, languages require the supporting technology of structural reflection, referential integrity and first class code.

The MPF interface, comprising a set of operations, is used to evolve a representation of the executing system. The program model available to the meta-program is a Hypercode graph. This is based on the Hypercode representation (Zirintsis, 2000) (Sec. 3.5) and extends it by representing the program code in a graph structure defined by the abstract syntax, instead of as flat text. It includes both program syntax and program data to capture program closure. The same representation is used for both evolving components and incorporating them into the new system.

The ability of a Hypercode graph to characterize program closure allows parts of a system to be represented after decomposition without losing their context. Component state is available for inspection by meta-programs and it may be preserved over evolution. New components may be defined using the current state without having to explicitly store and re-initialize values.

The MPF has the advantage over other systems that Hypercode can represent executing code. Since a Hypercode graph can represent closure it may be used as a representation for introspecting the executing system. The introspection allows a meta-program to view parts of a system. Any data or code value may be viewed as a Hypercode graph and the view always reflects the up-to-date value. The Hypercode graph is available at any stage of the evolutionary process and the meta-program performs evolutions by altering the graph and reflecting the changed graph into the executing system.

In Table 1.2, the MPF is evaluated against the categories for evolving systems defined earlier.

Table 1.2. The Meta-Programming Framework. This table categorizes the support for evolution in the MPF.

Dynamic Change

Openness	Extant and newly defined components can be introduced by specifying them as Hypercode graphs.
Maintain State	Hyperlinks are maintained over change to preserve internal state.
Connections Bound and Unbound	Composition binds connections at runtime and decomposition unbinds them.
Reflection	Language support for reflection.

Autonomous Change

Automation	The MPF provides an interface for automatic changes.
Automated Motivations for Change	Contained in management components and not part of the MPF itself.
Automated Support for Openness	New behaviours can be introduced automatically using reflective facilities in the programming language.
Model Available to Management Components	Hypercode graphs.

Change Localization

Higher Order Connectors	No
Programmable Connectors	No
Partial Decomposition	Yes using the decomposition operator.

Change Management

Change Management	The mechanisms provided by the MPF can be used as part of a change management system.
View of Executing System	Yes - a Hypercode graph representation of all values. Software architecture and implementation level views can both be constructued using the MPF.
Partial Views	Yes - all values are viewable as Hypercode.
Vocabulary for Specifying Changes	Changes can be specified in terms of MPF operations.
Open Adaptation	Yes - meta-programs can introduce new behaviours specified in terms of Hypercode graphs.
Closed Adaptation	Yes - meta-programs can introspect the current system.
Architectural Styles	No

1.8 Summary

This chapter begins by defining co-evolution as the process whereby software systems evolve in tandem with the business processes for which they are used. Co-evolution motivates the construction of systems that can evolve incrementally. A paradigm from control systems, feedback and feedforward controllers, is used to illustrate the difference between open and closed adaptive engines in evolving systems. Open adaptive engines evolve a system using feedback from environmental inputs and support the introduction of new behaviours. Their implementation is not well understood and they usually involve developers in the evolution process. In contrast, multiple implementations of closed adaptive engines exist, commonly using probes and gauges.

Software architectures are introduced, along with the advantages of approaching evolution from a high level of abstraction. Meta-programming is presented as a way of automatically manipulating executing systems, and therefore evolving them. The characteristics of evolving systems are depicted in terms of *when*, *what*, *how* and *where* systems are evolved and *who* drives the evolution. Following this, a set of categories are defined to allow comparison between different frameworks for evolution. Existing work is evaluated against these categories in the next chapter.

Finally, the Meta-Programming Framework is introduced as a mechanism to aid automatic, incremental evolutions. Its principle components are defined to be a representation of program closure and an interface that allows the representations to be evolved. This combination facilitates the incremental evolution of a running system. A table is used to classify the MPF with respect to the defined categories.

1.9 Contribution

This thesis proposes the hypothesis that the Meta-Programming Framework provides the mechanisms for automated evolution of software systems such that evolution proceeds with minimal disruption and is not restricted to any particular part of the system or type of evolution.

The development of software systems capable of evolving requires mechanisms to support the evolution process. Change Management Frameworks have been developed in order to manage software adaptation from a software architecture perspective. These frameworks consist of a system model in the form of a software architecture and a set

of constraints on that model to ensure it conforms to required behaviour. Components, usually probes and gauges, are applied to test the running system. The results they deliver are evaluated against the constraints. If it is determined that the constraints are violated then a repair strategy is used to change the system. The MPF can be integrated into a Change Management Framework, providing it with an interface to the executing system. In this structure, software architecture styles and evolution policies, such as timing and repair strategies, are taken care of by other components of the Change Management Framework.

A Change Management Framework that applies the MPF can use introspection to test the running system, an approach that is more powerful and flexible than probes and gauges. It can also use the MPF's mechanisms for unrestricted change that can be automated or include developer interaction. Repair strategies can be formulated using these mechanisms. In addition, the MPF interface is evolvable so it can be adapted to suit different systems and change management policies and evolve with the system over time.

The MPF provides a combination of technologies that support automatic and incremental evolution. Decomposition allows part of an executing system to be stopped. Hypercode graphs and the set of operations for their traversal, manipulation and evolution make up an interface to the decomposed part. Meta-programs, as part of a change management framework, use the interface to introspect and evolve the system. Evolved parts are introduced back into the executing system using structural reflection. Internal state in the evolved part of the system may be retained over the change, because Hypercode graphs capture program closure.

The MPF's combination of technologies for incremental evolution contributes the mechanisms that a change management framework needs to automate the evolution of an open system. Change management components can access a Hypercode graph representation of any value in the system. This can be used to evolve programs without losing state, introduce new behaviours and evolve the framework itself.

Chapter 2

Software Architectures and Frameworks for Evolution

This chapter is a review of work which has approached software evolution from the perspective of software architectures. The work discussed here is Darwin, Gerel, Weaves, ArchJava, Wright, ACME and Rainbow, ArchStudio and C2, and Intentional Programming. In each section, a table is shown which compares the system against the framework characteristics defined in Chapter 1.

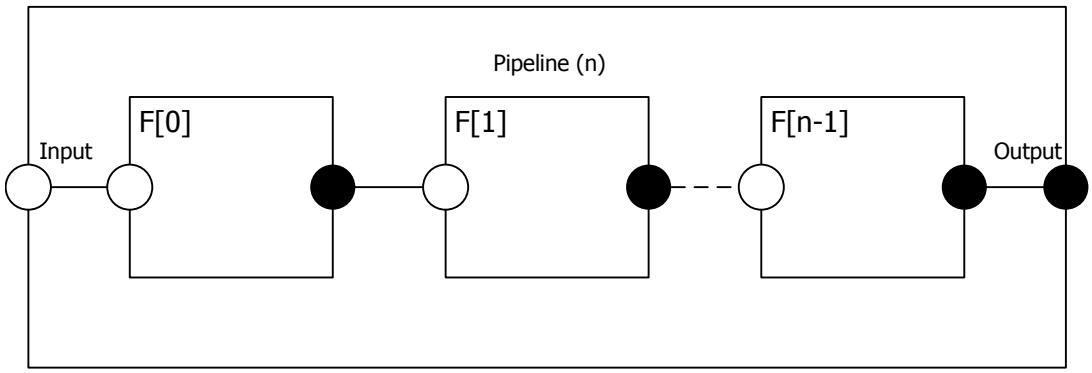
This chapter is a review of work which has approached software evolution from the perspective of software architectures. The work discussed here is Darwin, Gerel, Weaves, ArchJava, Wright, ACME and Rainbow, ArchStudio and C2, and Intentional Programming. In each section, a table is shown which compares the system against the framework characteristics defined in Chapter 1.

2.1 Darwin

Darwin (Magee et al., 1995; Magee et al., 1994; Magee and Kramer, 1996a; Kramer and Magee, 1990) is an ADL developed for the design and specification of distributed architectures. Its operational semantics are based on the monadic π-Calculus (Milner, 1999), to facilitate modelling of mobile communicating processes. Darwin has been specifically developed as a declarative language, in order to provide a clear separation of concerns and support automated reasoning about structural concepts.

Darwin architectures are defined as hierarchic compositions of interconnected components. A system consists of a set of component types that have multiple instantiations. *Composite* component types are constructed from other components. *Primitive* component types have a behaviour and do not contain other components. Component interaction is represented by bindings between the services *required* and *provided* by components through their *ports*. A component may provide a service to many components that require it, but a required service can only be connected to one provider. Provided services may only be bound to required services and vice versa and *binding* a provided and required service port connects them directly.

An example pipeline component is shown in Figure 2.1 (Magee et al., 1995). The pipeline is a composite component constructed from n primitive filter components. Filter components each have one required (input) and one provided (output) service of type *stream*, shown in the diagram as white and black circles respectively. The pipeline component instantiates an array of filter instances and binds the output service of each filter to its successor's input. The instantiation is defined by the keyword **inst**. Binding takes place in the **bind** clause between services connected by –, as in *F[k+1].input – F[k].output*. Because components may not communicate directly outside of their containing component, the first and last filters in the pipeline are connected to the pipeline's input and output services.




```
component filter {
   provide output: stream;
   require input stream;
}

component pipeline(int n) {
   provide output;
   require input;

   array F[n]: filter;
   forall k: 0..n-1 {
      inst F[k];
      when k < n-1;
         bind F[k+1].input -- F[k].output;
   }
   bind
      F[0].input -- input;
      output -- F[n-1].output;
}
```

Figure 2.1. Darwin. Filter and pipeline components (Magee et al., 1995).

2.1.1 Dynamic Behaviour

Support for evolution in Darwin is provided in the form of *lazy* and *direct dynamic* instantiation. Lazily instantiated components are statically defined types that are only instantiated when needed, that is, when their services are accessed by another component. This allows a system to create new components according to a fixed pattern. Using lazy instantiation, components are dynamically connected, viz., their context is defined dynamically. In contrast, the context of components created through direct dynamic instantiation must be statically defined and it is the component type that is defined dynamically.

Lazy Instantiation Example

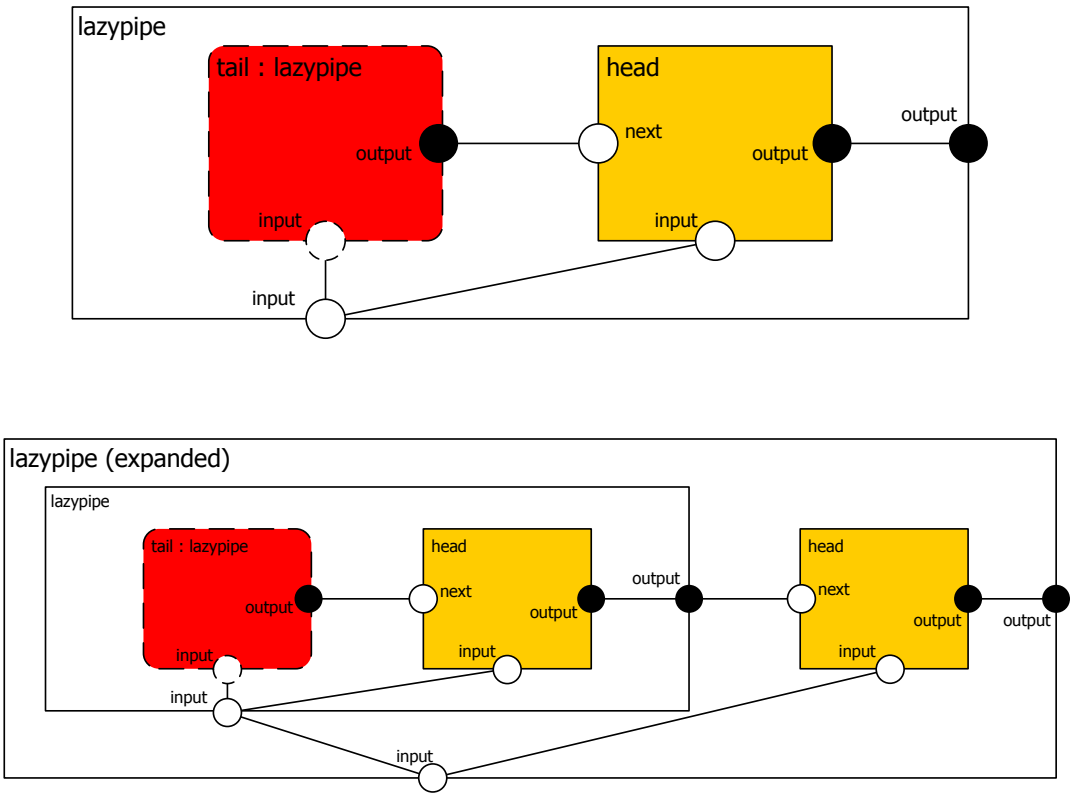
A pipeline that grows through lazy instantiation is shown in Fig. 2.2 (Georgiadis, 2002). The top part of the figure shows a *lazypipe*, which expands into the pipeline shown in the centre of the figure. The expansion takes place as the *head* component requires services from the *tail* component, causing a new *lazypipe* be instantiated and extending the tail. Note that the pipeline is extended as the tail lazypipe expands to a head and a new tail, causing the pipeline to have increasing numbers of head components and only one tail component.

Lazy instantiation happens only when a service user tries to access the service. In this example, a new *lazypipe* is instantiated, adding a filter to the pipeline, when *head.next* attempts to use a service provided by *tail.output*. When a *lazypipe* receives input, it is sent to the *head* component. *head* then requires input from *tail*, causing *tail*, which is of type *lazypipe*, to be instantiated.

The Darwin code at the bottom of the figure shows how a *lazypipe* is defined. It requires input and provides output. A filter component (*head*) is instantiated using the keyword **inst**. The *tail* component's instantiation is prefixed with the keyword **dyn**, meaning it will be lazily instantiated. The final part of the component's code connects the various provided and required services in the *lazypipe*.

Direct Dynamic Instantiation Example

Components created using *direct dynamic* instantiation have dynamically introduced types. However, the services they will be connected to are required to be statically defined. Whereas lazy instantiation only allows the creation of new component instances, direct dynamic instantiation supports the introduction of new functionality at




```
component lazypipe {
  require input;
  provide output;

  inst
    head: filter;
    tail: dyn lazypipe;
  bind
    input -- head.input;
    head.output -- output;
    head.next -- tail.output;
    tail.input -- input;
}
```

Figure 2.2. Darwin. Lazy instantiation (Georgiadis, 2002).

runtime. Component types for direct dynamic instantiation are defined in a meta-level configuration, essentially a Darwin script, and instantiation is driven by interpreting this script at runtime. Component instances created using direct dynamic instantiation are anonymous. In order to refer to services they provide, service references can be sent in messages to form bindings dynamically.

Figure 2.3 (Georgiadis, 2002) depicts a client / server system that expands using direct dynamic instantiation when the server component creates new client components. New clients are created by the server's *new* port as required by the server's internal computations. Each client component's type is dynamically defined, but its interface will always connect to *context.input* and *server.comm*.

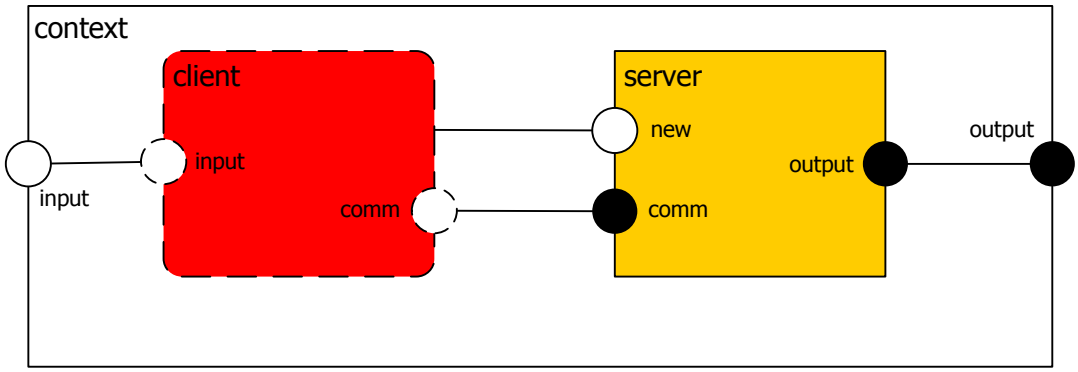
The Darwin code for this example defines the interfaces for the *client* and *server* components in lines 1-7. The server's required port *new* is typed as <*dyn*> (line 5), an implementation dependent type defining components that can be invoked dynamically. In the *context* component, a *server* is instantiated in line 11. When it is bound in the *context* component's **bind** clause, its *new* port is used to create a dynamic instantiation of a *client* component using the keyword **dyn** (line 14). Syntactically, direct dynamic instantiation takes place in the **bind** clause, whereas lazy instantiation occurs in the **inst** clause.

In summary, lazy instantiation allows new component instances to be added to a software architecture. Direct dynamic instantiation allows the inclusion of new component types, thereby adding new functionality.

2.1.2 Self-Organizing Systems based on Darwin

Darwin components have been shown to be suitable for describing self-organizing systems. One example of their use is as part of an environment for distributed configuration management that reconfigures a system at runtime (Fossa and Sloman, 1996).

Constraints are used to restrict self-organizing systems so that they remain well formed with respect to their configuration despite re-organizations, i.e. architectural properties are preserved (Magee and Kramer, 1996b). They have been used as the basis for a self-organizing architecture (Georgiadis et al., 2002; Georgiadis, 2002) expressed in Alloy (Jackson, 1999) but modelled on Darwin components. Alloy is an ADL that




```
1  component client {
2    require input, comm;
3  }
4   component server {
5    require new <dyn>;
6    provide comm, output;
7  }

8  component context {
9    require input;
10   provide output;
11   inst s: server
12   bind
13      output -- s.output;
14      s.new -- dyn client;
15      s.comm -- client.comm;
16      client.input -- input;
17 }
```

Figure 2.3. Darwin. Direct dynamic instantiation (Georgiadis, 2002).

has been used because of its automated analysis tools. The work has focused on distributed systems where components can fail arbitrarily, causing the architecture to be re-organized accordingly.

Each component contains its implementation, a component manager and a configuration view. The component manager contains a set of constraints, used as the basis for deciding when and what reconfigurations to make. When a component manager makes a change to its component's bindings it updates the configuration view, which is a current depiction of the entire system maintained in the component. It also broadcasts the change to other component managers.

Constraints can be either local or global and are used to specify both structural and evolutionary aspects of the architecture. Local constraints, for example, the constraint that each component should have one provided and one required port, can be checked by an individual component. This is done using Alloy tools (Jackson, 1999) to perform an automated analysis. If a local constraint is broken, the component can perform a set of actions to repair itself. Global constraints, for example, that a pipeline architecture does not form a circle, are harder to enforce than local constraints and require actions that are part of a distributed algorithm. One of the goals of the work is to derive the action rules, currently a manual task, automatically from the set of constraints.

In Table 2.1, Darwin is compared against the categories defined in Table 1.1.

Table 2.1. Darwin. This table categorizes the support for evolution in Darwin.

Dynamic Change

Openness	Yes - using direct dynamic instantiation.
Maintain State	Continuity - No
Connections Bound and Unbound	Components can be bound but not unbound at runtime because Darwin is declarative. Unbinding is intended to be part of the implementation framework.
Reflection	Reflection not in the Darwin ADL. Intended as part of the implementation framework.

Autonomous Change

Automation	Yes - in the work on self-organizing systems.
Automated Motivations for Change	Constraints determine what changes to make and when to make them.
Automated Support for Openness	No
Model Available to Management Components	Configuration managers maintain an up-to-date view of the entire system for each component.

Change Localization

Higher Order Connectors	No - component ports are directly connected to each other.
Programmable Connectors	No
Partial Decomposition	Darwin permits runtime change to components which have reached their reduction limit, described as quiescent. Essentially, this means they are not communicating with other components.

Change Management

Change Management	In the work on self-organizing systems each component has its own change manager.
View of Executing System	Yes - each component has its own up-to-date view of the system.
Partial Views	No
Vocabulary for Specifying Changes	Alloy rules
Open and Closed Adaptation	Yes - Darwin is able to describe systems which perform reconfigurations as well as introduce new behaviours and can therefore represent both open and closed adaptive engines.
Architectural Styles	Uses Alloy

2.2 Gerel

Gerel (Endler and Wei, 1992) is a generic reconfiguration language and was one of the earlier languages to incorporate dynamism at the level of software architectures. It aims to support both ad-hoc and programmed changes to an architecture. Ad-hoc changes alter functionality and are applied interactively. Programmed changes are planned at development time and performed automatically. The programmed changes are atomic and do not occur concurrently with ad-hoc changes to avoid leaving the system in an inconsistent state. They are also generic in order to be robust to the system evolution resulting from ad-hoc changes. The genericity of programmed changes is a result of the dynamic selection of objects, over which the changes will operate, according to their structural properties. Component instances are not explicitly named in the change script. Programmed changes are only applied when a set of preconditions is satisfied by the current configuration.

All programmed changes are restricted to the component in which the changes are defined and affect the system's connectivity structure. Components can be added and removed, connected and disconnected. Structures are defined in terms of component types and instances, portsets and ports. The approach supports a separation of structural reconfiguration and functional application programming concerns.

A system defined in Gerel consists of *program* and *configuration* type components. Program components contain the application's functionality and can be defined in any programming language as long as they implement a common interface for ports. Configuration components are defined in Gerel. They create instances of component types and bindings between them. Configuration types also have a port interface. A configuration component may contain instances of other configuration components, forming a hierarchical system.

The component types for an example client / server are shown in Figure 2.4 (adapted from Endler and Wei (1992)). At the top left of the figure (1), an interface for communication between the client and server components is defined. The *client_server_interface* definition uses an interface specification language and defines a set of ports and the data type of the messages that can be sent on those ports. Ports are defined using the keyword **inport** for input channels and **outport** for output channels. The type *io_ports* defines a *portset* consisting of an input and and output channel. The *in* channel receives a signal using asynchronous communication. The *out* channel uses synchronous communication, sending something of type *message* and waiting on a **result** *signal*.

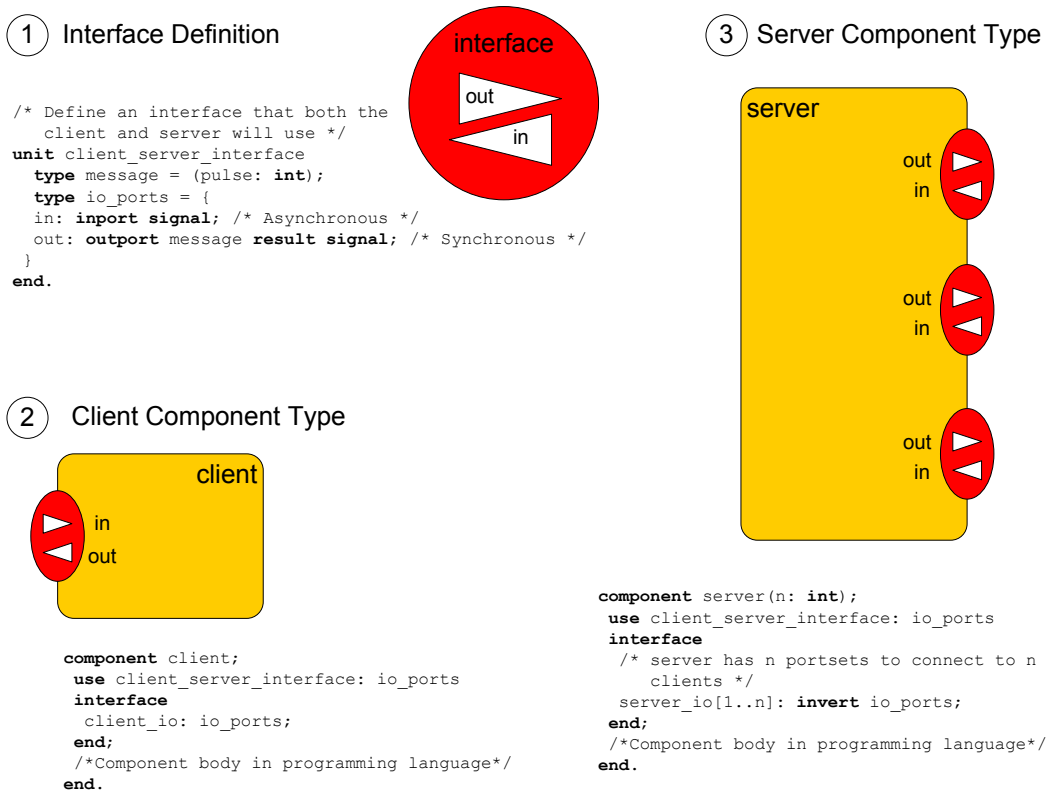


Figure 2.4. Gerel. Component types for a client / server system in Gerel.

The client component (2) is defined to **use** the *io_ports* portset in the *client_server_interface*. In the client's **interface** one portset of type *io_ports* is instantiated and called *client_io*. In the definition of the server component (3), the same interface is used, but n instances of the portset are instantiated so that the server can communicate with n clients. It also *inverts* the portset, reversing the directions of the communication channels. The result of inverting the portset is that an *in* port on the client has the same type definition as an *out* port on the server.

The client and server are program component types. A configuration component is used to instantiate server and client components and bind them together in a system illustrated in Figure 2.5. The code for the client / server system (Fig. 2.6) uses the client and server component types defined earlier (line 2). Three clients and two servers are instantiated (lines 3-4). Each server has three *server_io* portsets and can therefore connect to three different clients. The second server is redundant and can take over if the first one fails. Clients are connected to the portsets of the first server using the *bind*

statement. A programmed change is included in the component (line 8) to disconnect the clients from *s[1]* and reconnect them to *s[2]*. The change does not take place until it is invoked by a server.

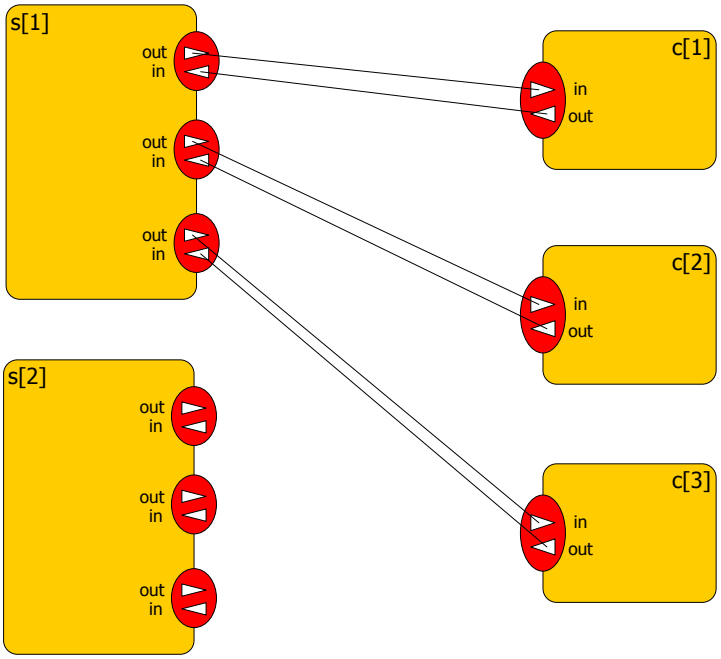


Figure 2.5. Gerel. The client / server configuration with two servers and three clients. On invocation of the change script clients are disconnected from s[1] and reconnected to s[2].

```
1  component system;
2      use client, server;
3      iterate i in [1..3] do create c[i] of client; end;
4       iterate j in [1..2] do create s[j] of server(3); end;
5      bind bind c[1].client_io s[1].server_io[1]
6      bind bind c[2].client_io s[1].server_io[2]
7      bind bind c[3].client_io s[1].server_io[3]
8      change relinkclients; /* a programmed change */ end.
9  end.
```

Figure 2.6. Gerel. The configuration component for the client / server system.

A programmed change consists of:

- A precondition for the current configuration, i.e., a structural property that must hold true before the change is executed.
- Structural properties of components, which can be selected to be changed.
- The change to be executed.

The preconditions for the change and the selection formulas are defined in Gerel-SL, a typed first order logic language.

The programmed change *relinkclients* (Fig. 2.7) will disconnect any number of clients from a server and reconnect each client to another server with a free port. In line three, an inbuilt Gerel-SL function is used to check that the change script is being invoked by a component of type server, meaning that clients cannot initiate their own reconnection. In lines five to eight, all clients *j* attached to the invoking server *k* are unbound and then rebound to a server *i* with a free portset. The expressions *client_portset* and *free_portset* are defined by separate Gerel-SL formulae. *s_linked* is an inbuilt function.

```
1  change relinkclients;
2     symbol i, j, k: portset io_ports;
3     condition c_type(invoker())=server;
4     execute
5     forall  j: client_portset(j) do
6        select k: s_linked(k, j) do unbind j k end;
7        select i: free_portset(i) do bind i j end;
8     end;
9  end.
```

Figure 2.7. Gerel. A programmed change script to unlink all the clients from a server and attach them to a new server.

In Table 2.2, Gerel is compared against the categories defined in Table 1.1.

Table 2.2. Gerel. This table categorizes the support for evolution in Gerel.

Dynamic Change

Openness	Ad-hoc changes can introduce new behaviours.
Maintain State	No
Connections Bound and Unbound	Yes - using the *bind* and *unbind* operations for connections.
Reflection	No

Autonomous Change

Automation	Yes - programmed changes operate automatically.
Automated Motivations for Change	Change scripts are executed when a structural precondition evaluates to true.
Automated Support for Openness	No - ad-hoc changes are not automatic.
Model Available to Management Components	No

Change Localization

Higher Order Connectors	No
Programmable Connectors	No
Partial Decomposition	Changes occur at runtime and are atomic so the system is not left in an inconsistent state but there is no decomposition mechanism.

Change Management

Change Management	Concerns of structural and functional reconfiguration are separated and the serialization of reconfigurations is managed by a system process.
View of Executing System	No
Partial Views	No
Vocabulary for Specifying Changes	Gerel and Gerel-SL
Open Adaptation	Yes - defined by ad-hoc changes.
Closed Adaptation	Yes - defined by programmed changes.
Architectural Styles	Gerel has constraints but they are not explicit.

2.3 Weaves

Weaves (Gorlick and Razouk, 1991) are networks of concurrently executing components that communicate by passing objects. They are suited to real-time systems or those which process large amounts of data and are focused on high performance flexible connectors. In a weave architecture, components are referred to as *tool fragments*, which send messages to each other via read and write ports. Tool fragments may themselves be weaves, in which case they are referred to as *subweaves*. Ports are connected by *queues*. The structure supports transparent, dynamic reconfiguration and insertion of probes and subweaves.

An innovation in weaves is to separate system observation from data capture, now commonly done using probes and gauges. Component boundaries are exploited in weaves as convenient sites for gathering information. Probes are low-overhead and can be inserted into the weave automatically to perform both debugging and analysis tasks. *Observers* behave like gauges. They collect information from probes and may query the weave with a regular frequency or be triggered by an external component. Changes to the weave, possibly instigated because of information gathered by observers, are effected by *Actors*. They translate change requests into low-level commands and apply them to the tool fragments and queues in the weave. For instance, an actor may suspend or resume a tool fragment's thread. Actors and observers were used in the implementation of weaves to build a tool to provide an animation of a weave. Each tool fragment and queue has:

- Its own observer that displays its current state and connectivity.
- Its own actor, which suspends or resumes it if the user clicks on the tool fragment in the display.

Figure 2.8 shows a diagram of a simple weave. Tool fragments communicate via read and write ports, which are in turn connected by queues. Tool fragments are constructed with three parts:

- A vector of arguments that includes read and write ports.
- A routine implementing the tool's behaviour.
- A thread to execute the behaviour.

Wrappers are used to include tool fragments that have not been specifically implemented as weave components.

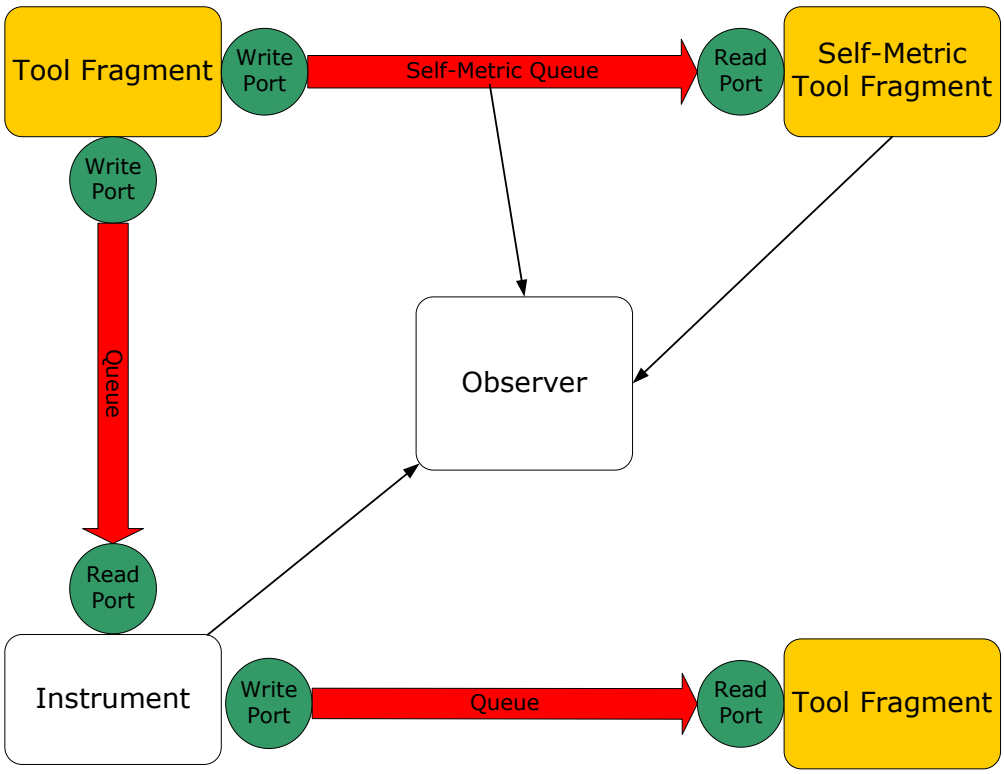


Figure 2.8. Weaves. The components of a weave.

Queues are finite buffers and pass values by reference. They can be defined to implement the queue operations optimally for a particular component connection. For instance, they may be synchronous or asynchronous and have multiple readers and writers. Concurrent access is allowed and must be managed by the tool fragments individually.

Observers gather information from probes in different parts of the weave. *Self-metric* queues and tool fragments contain built in probes that may be queried by the observer. In addition, data is collected from *instruments*, which are transparent components placed between tool fragments to monitor data flow. They are connected to the tool fragments with queues and do not interfere with the weave's execution. For

example, an instrument may count the number of objects passing through it, or count the number of objects of a certain type.

The runtime behaviour of a weave may be modified by changing its topology, that is, altering the connections between the tool fragments. In order to support incremental construction, tool fragments do not know what they are connected to, or the semantics of other tool fragments. New sub-weaves can be attached to weaves and tool fragments can be replaced, as long as the new tool fragment has the same connectivity interface.

A weave implementation may be constructed manually, interactively using a graphical editor, or generated from a set of high level goals and tool descriptions by a *weaver*. The software architecture is directly represented in the weave implementation. Tool fragments and queues are implemented as C++ objects, which present methods allowing them to be manipulated to, for example, alter the weave topology.

In Table 2.3, Weaves is compared against the categories defined in Table 1.1.

Table 2.3. Weaves. This table categorizes the support for evolution in Weaves.

Dynamic Change

Openness	Yes - weaves are designed to be constructed incrementally.
Maintain State	Components do not retain state over a change but when a component is detached from a *queue* the *queue* will maintain its internal state and continue transmitting to the next component it is connected to.
Connections Bound and Unbound	Yes - elements in a Weave do not make assumptions about what they are connected to, allowing changes in the topology to be applied dynamically.
Reflection	No

Autonomous Change

Automation	Actors automatically apply change requests to the weave.
Automated Motivations for Change	Information gathered by observers provides a motivation for change.
Automated Support for Openness	Yes because actors can automatically attach new behaviours to the weave.
Model Available to Management Components	No

Change Localization

Higher Order Connectors	Queues are higher order connectors.
Programmable Connectors	Yes
Partial Decomposition	Yes - a tool fragment can be isolated by disconnecting its ports and then stopped by an actor.

Change Management

Change Management	No but explicit connectors aid reasoning about and understanding of communication.
View of Executing System	No
Partial Views	No
Vocabulary for Specifying Changes	Tool fragments provide an interface for *actors*. The minimum interface is thread suspension and reactivation.
Open Adaptation	No
Closed Adaptation	There is no adaptive engine but observers and actors provide the mechanisms to gather information and apply changes.
Architectural Styles	No

2.4 ArchJava

ArchJava is a set of extensions to the Java programming language that integrates a software architecture specification into the Java source code (Aldrich et al., 2002a; Aldrich et al., 2002b; Aldrich et al., 2002c; Aldrich, 2003; Aldrich, 2005 (Submitted for publication)). A tool is used to automatically generate a visualization of the software architecture from the ArchJava implementation code. The ArchJava compiler statically checks that the program conforms to the architecture by ensuring that communication and data sharing only take place where they are explicitly allowed. This is referred to as *communication integrity*.

The problem of ensuring communication integrity is broken down into two parts: inter-object method calls and data sharing. Objects should only be allowed to call methods on other objects where it is explicitly allowed by the architecture and data may only be shared between objects when explicit permission is given.

2.4.1 Restricting Inter-Object Communication

To control inter-object method calls, ArchJava allows the specification of components, connections and ports by extending the Java language. A component can only communicate with other components to which it is connected through its ports. A port specifies which methods it *provides* and *requires*, as in Darwin. No other method calls between components are allowed. A component's *required* methods are called in the component and defined in another component. Its *provided* methods are implemented in the component and called by other components. A *required* port in one component is attached to a *provided* port in another using the keyword *connect*. Although not part of the language, ArchJava has been extended with first class connectors (Aldrich et al., 2003). To guarantee communication integrity the type system ensures that components can only call each other's methods via ports. There is no way of breaking a connection between ArchJava components. They will stay connected until they are no longer reachable and are therefore garbage collected.

Software architectures in ArchJava are strictly hierarchical: a component may not be shared by two containing components. A component can only call the methods of a component which is its sub-component and only components at the same architectural level can be connected using the ArchJava *connect* construct. Communication which crosses component boundaries at a higher architectural level is not allowed.

This means that sub-components may not communicate outside their enclosing component.

2.4.2 Restricting Data Sharing

The second part of ensuring communication integrity is constraining data sharing between objects. This is done using another extension to Java called AliasJava, which integrates with ArchJava. In AliasJava objects are assigned an *ownership domain*. Objects residing outside an ownership domain require explicit permission to access the objects within. Permission is granted by *linking* one domain to another, allowing the objects in the first domain to reference objects in the second. Ownership domains are hierarchical and an object has permission to access objects in the domains that it declares as well as other objects in the same domain.

Ownership domains are not suitable for modelling some data sharing behaviours, for example, an object being passed along a pipeline when no persistent reference to the object is created. In order to handle these behaviours, ownership domains are extended to be able to define objects as either *unique* or *lent*. Unique objects may be passed between ownership domains as long as there is only ever one reference to them. Lent objects may be used outside their ownership domain but no persistent references to them may be created.

The top level ownership domain *shared* contains all objects. Figure 2.9 illustrates the relationship between ownership domains and objects. *myObject* is inside the top-level *shared* ownership domain. The objects nested inside *myObject* are the objects declared there. The domain *owned* inside *myObject* is a private domain contained in every object by default. In addition, *myObject* declares the public domain *accessible*, whose objects may be accessed from outside *myObject*. Creating a *link* from the *accessible* domain to the *owned* domain allows objects declared inside *accessible* to access objects in *owned*.

2.4.3 Example Software Architecture

In ArchJava, the component model and ownership domains are combined. The component hierarchy uses ownership domains and domains may be shared between components by passing them along connections. An example of a pipeline architecture

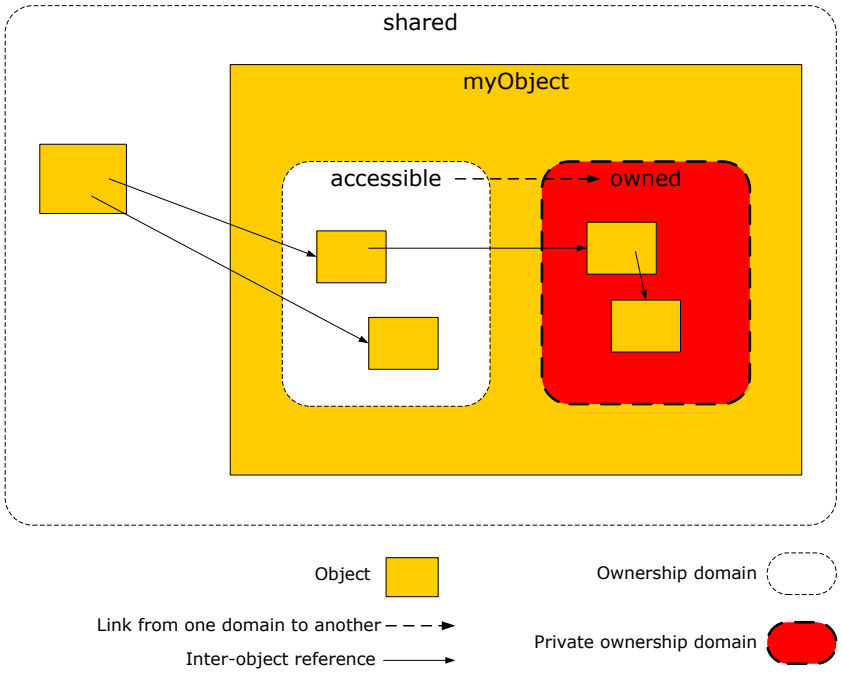
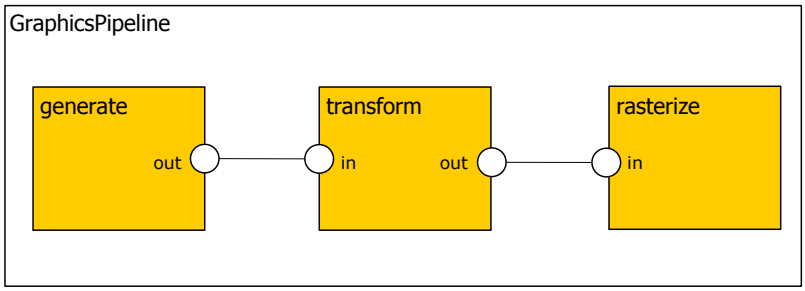


Figure 2.9. ArchJava. Ownership Domains - adapted from (Aldrich, 2005 (Submitted for publication)).

(Fig. 2.10) illustrates ownership domains in the component hierarchy. In this example, components in the pipeline cannot access objects concurrently, but they can share objects by passing them along the pipeline.

The graphics pipeline in Figure 2.10 has three sub-components in it, to generate, transform and rasterize images. The ArchJava code in the example first defines a *GraphicsPipeline* class and then defines one of its sub-components - *Transform*. The other two sub-components are not shown. The classes in this example are defined using the **component** keyword, but classes may also be declared that are not components.

The *GraphicsPipeline* class (line 1) defines three sub-components and their topology. It instantiates three objects to form the pipeline: *generate* (line 2), *transform* (line 3), and *rasterize* (line 4). Declaring these as **owned** means they are sub-components of *GraphicsPipeline* and contained in its default private domain, therefore they may not be referenced by objects outside *GraphicsPipeline*. The next part of *GraphicsPipeline* (lines 5 - 6) defines a communication pattern to which sub-components must conform when they create connections at runtime. The pattern forms the pipeline shown in the diagram. The keywords **connect pattern** indicate that the *out* port of a *Generate* type object may




```
1  public component class GraphicsPipeline {
2      protected owned Generate generate = ...;
3      protected owned Transform transform = ...;
4      protected owned Rasterize rasterize = ...;

5      connect pattern Generate.out, Transform.in;
6      connect pattern Transform.out, Rasterize.in;

7      public GraphicsPipeline() {
8          connect(generate.out, transform,in);
9          connect(transform.out, rasterize.in);
10     }
11 }

12 public component class Transform {
13     protected owned Trans3D currentTransform;

14     public port in {
15         provides void draw( unique Shape s);
16     }
17     public port out {
18         requires void draw( unique Shape s);
19     }

20     void draw( unique Shape s) {
21         currentTransform.apply(s);
22         out.draw(s);
23     }
24 }
```

Figure 2.10. ArchJava. The GraphicsPipeline architecture (Aldrich, 2005 (Submitted for publication)).

connect to the *in* port of a *Transform* type object. The ports of the *generate*, *transform* and *rasterize* component objects are actually connected in the *GraphicsPipeline* constructor (lines 8 - 9). The ArchJava compiler will check that these connections conform to the connect patterns in lines 5 and 6.

The source code for the *Transform* class is shown in the second part of the figure (line 12). The class definition contains the types of its ports and implements its provided method. Ports are declared with provided (line 15) and required (line 18) methods. The *in* port provides a method *draw*, which may be called by other components and passed a *Shape*. The keyword **unique** is used to ensure that the *Shape* object being sent along the pipeline is not accessed concurrently by two components. Methods provided on a port must be implemented inside the component, therefore *Transform* implements the *draw* method. The definition of *draw* (line 20) applies a transformation and then sends the *Shape* to the next component in the pipeline by sending a message to the required method *draw* on the *out* port.

In Table 2.4, ArchJava is compared against the categories defined in Table 1.1.

Table 2.4. ArchJava. This table categorizes the support for evolution in ArchJava.

Dynamic Change

Openness	New components may be instantiated at runtime but there is not support for new behaviours.
Maintain State	No
Connections Bound and Unbound	Components may be dynamically connected but not disconnected.
Reflection	Java reflection mechanisms are available.

Autonomous Change

Automation	No
Automated Motivations for Change	No
Automated Support for Openness	No
Model Available to Management Components	The software architecture is explicit in the ArchJava source code and ArchJava has classes to reify architectural constructs.

Change Localization

Higher Order Connectors	No - components are connected directly through their ports but the language may be extended to include first class connectors.
Programmable Connectors	No
Partial Decomposition	No

Change Management

Change Management	No - and in expressing the software architecture in the implementation language, ArchJava loses the separation of architecture and implementation.
View of Executing System	Yes - tools generate visualizations of the software architecture.
Partial Views	No
Vocabulary for Specifying Changes	No
Open Adaptation	No
Closed Adaptation	No
Architectural Styles	No

2.5 Wright, ACME and Rainbow

2.5.1 Wright

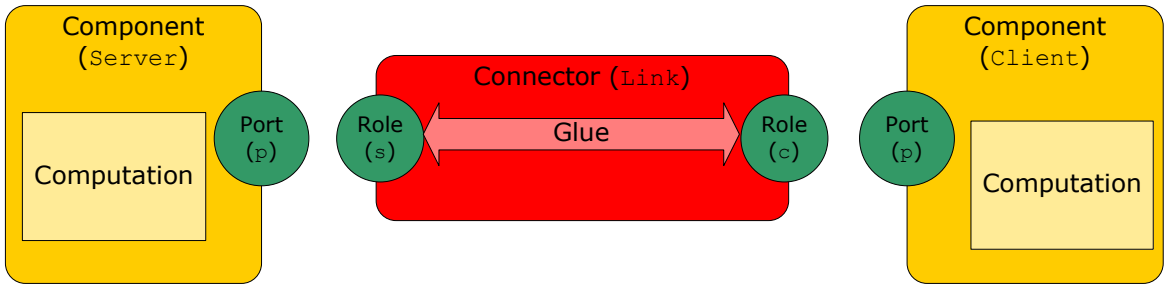
Wright (Allen, 1997; Allen and Garlan, 1997; Allen et al., 1998) is an ADL formally based on CSP (Hoare, 1985), that supports consistency checking with standard model checkers. One of its defining characteristics is that connectors are distinguished as first class entities. It is argued that, although connectors can be modelled as components, having them as separate entities directly supports the abstractions used by software architects in their designs as well as re-use.

A Wright component has *ports* to characterize its interactions and a *computation* to specify its functionality. A connector has *roles* that map to components' ports, and *glue* to define its behaviour. For example, a connector in a client / server system could have a role for the server and a role for the client. Its glue would co-ordinate the roles of the client and the server. Components and connector instances joined by *attachments* form *configurations*.

Component and connector types are defined before being instantiated and attached in a configuration. The types of elements for a client / server style system are defined in Figure 2.11. There is a *Client* component, a *Server* component and a connector called *Link*. These types, in addition to a set of constraints, form an architectural style.

In the example, it can be seen that components are described in terms of their interfaces, or ports, and their internal *computation*, which is defined using CSP-like notation. The *Client* component (line 2), for example, has one port *p* (line 3), which either sends a request ($\overline{request}$), receives a *reply* and starts again, or else it finishes (indicated by §). The choice to finish is made internally by the Client component, shown by the internal choice operator $\sqcap$. Wright has two choice operators: $\sqcap$ and $\Box$, which indicate that a choice is determined internally or externally respectively. For example, in the former case a component stops itself and in the latter case the component is stopped by another component.

The *Server* component (line 5) receives a request on its port before sending a reply. In the *Server*, the choice to finish is determined externally, meaning that the server will not terminate until the client is ready. The **Connector** (line 8), which will link up the client and server, is defined using **Role**s (lines 9 - 10) to characterize its interface and **Glue** (line 11), which co-ordinates the activities between roles. *Link* has a role for the




```
1  Style Client-Server
2    Component Client
3      Port p = request → reply → p ⊓ §
4      Computation = internalCompute → p.request → p.reply → Computation ⊓ §

5    Component Server
6      Port p = request → reply → p □ §
7      Computation = p.request → internalCompute → p.reply → Computation □ §

8    Connector Link
9      Role c = request → reply → c ⊓ §
10     Role s = request → reply → s □ §
11     Glue = c.request → s.request → Glue
12               □ s.reply → c.reply → Glue
13               □ §

14   Constraints
15   ∃!s ∈ Component, ∀ c ∈ Component : TypeServer(s) ∧ TypeClient(c) ⇒ connected(c,s)
16 EndStyle
```

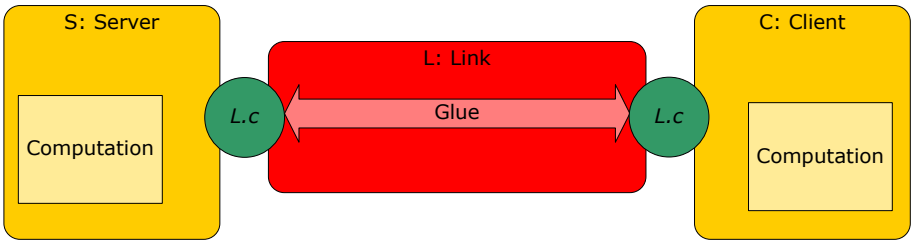
Figure 2.11. Wright. Component type definitions for a client / server style system (Allen et al., 1998).

client (line 9) and for the server (line 10). In the concrete system the connector's ports will be instantiated as the connector's roles, the definitions of which match. The glue in the connector definition can handle:

- A request from a client, which is sent on to the server.
- Or a reply from the server that is forwarded to the client.
- Or it can finish.

The **Constraints** on the component and connector types restrict the architecture so that there is only one server component (*exists*!s) and every client is connected to it.

The configuration of a client server system is shown in Figure 2.12. A *Server* component, a *Link* connector and a *Client* component are instantiated. In the *attachments* part of the configuration the client and server ports are instantiated as their respective connector roles.



Configuration
Style `Client-Server`
Instances `S: Server; L: Link; C: Client`
Attachments `S.p` **as** `L.s; C.p` **as** `L.c`
EndConfiguration

Figure 2.12. Wright. Instantiating and attaching component types in a configuration (Allen et al., 1998). This system uses the types defined in Figure 2.11.

To incorporate dynamism, Wright has been extended to incorporate the definition and analysis of architectural reconfiguration. The system is initially defined as a set of architectures, each with a different topology. Dynamic behaviour involves swapping

between these predefined topologies. Explicit points are identified in the protocols defining components and connectors at which reconfiguration, or *control*, events can take place. The approach aids change management by separating reconfiguration behaviour from the functional behaviour of the application. Reconfiguration behaviour is contained in a *configuror*, which defines a number of topologies for the system and a protocol that decides when to switch between them.

The Aesop toolkit (Garlan et al., 1994) complements Wright by supporting the generation of development environments customized for architectural styles and the integration of tools for formal analysis.

2.5.2 ACME

ACME was developed as an interchange language for ADLs (Garlan et al., 1997). It aims to support language independent formal analysis of software architecture and software architecture tool integration. Elements of Wright used in ACME include components, ports, connectors, roles and attachments. In addition, ACME uses templates to represent recurring systems, e.g., a client template would be defined for a client / server system. Styles are represented by sets of templates. To allow translation into ACME from more specialized ADLs, an ACME description can be annotated with *properties* that include any extra information.

2.5.3 Armani

Armani is an ADL with support for specifying styles (Monroe, 1998). Its core elements are in the same vein as those of ACME and Wright. In addition, it includes constructs for:

Design vocabulary The set of element and property types in the system.

Design rules Rules to specify invariants, heuristics and composition constraints that can be used to analyse architectures.

Architectural styles A collection of all the design elements for a system.

These correspond to the elements of a style described in Section 1.2.1.

2.5.4 Rainbow

In the Rainbow framework, self-adapting software has been implemented using style constraints on a software architecture model of the executing system to drive changes (Garlan et al., 2004; Garlan and Schmerl, 2002; Schmerl and Garlan, 2002; Garlan et al., 2003). Managing adaptation at the software architecture level enhances understanding and re-use of change policies and supports reasoning about the system as a whole. The executing system can be analysed according to constraints on the architectural style to establish when repairs are necessary. Repair strategies can be formulated in terms of the software architecture, supporting system changes at a global level. Architectural styles are used to determine what to measure in the executing system, what constraints to evaluate, what to do when a constraint is violated, and how to repair the system.

The architecture model has a style defined as a set of component and connector types, constraints, properties and analyses. Constraints restrict the interactions allowed between elements. Properties are attributes associated with particular elements. Analyses return information about a system, such as whether it meets performance requirements. Associated with each style are *adaptation operators* and *adaptation strategies*. Adaptation operators define changes to a style instance, which may include changes to component properties. Generic examples are operators to add or remove components, but style specific operators may also be defined. Adaptation strategies determine the cause of a constraint violation in the implementation and how to remedy it.

Keeping an external model of the software architecture introduces the need for mechanisms to bridge the gap between the model and the implementation. Rainbow has mechanisms for both monitoring and changing the system.

Rainbow is a generic runtime framework to support runtime change based on an external architectural model, which can be adapted to suit particular styles and properties. The components of the framework are shown in Figure 2.13. Information about the executing system is gathered by *probes* and a tool for *resource discovery*, which detects new elements in the system. These report to *gauges* in the *Architecture Layer*. A *Translation Infrastructure* in-between maps implementation constructs to architecture level constructs and vice versa. Relevant information from gauges is used to update the properties in the *architecture model manager*. When a property is changed, the *constraint evaluator* checks the new model with respect to the constraints. If it is determined that a constraint has been violated, the *adaptation engine* is called. It signals the architecture model to suspend the monitoring mechanisms and takes a snapshot of its current

state. It then determines the adaptations necessary to repair the violation and these are carried out by the *adaptation executor*, which uses the interface proffered by the *effectors* to apply them to the executing system. After an adaptation the constraints are re-evaluated to determine whether the repair was effective.

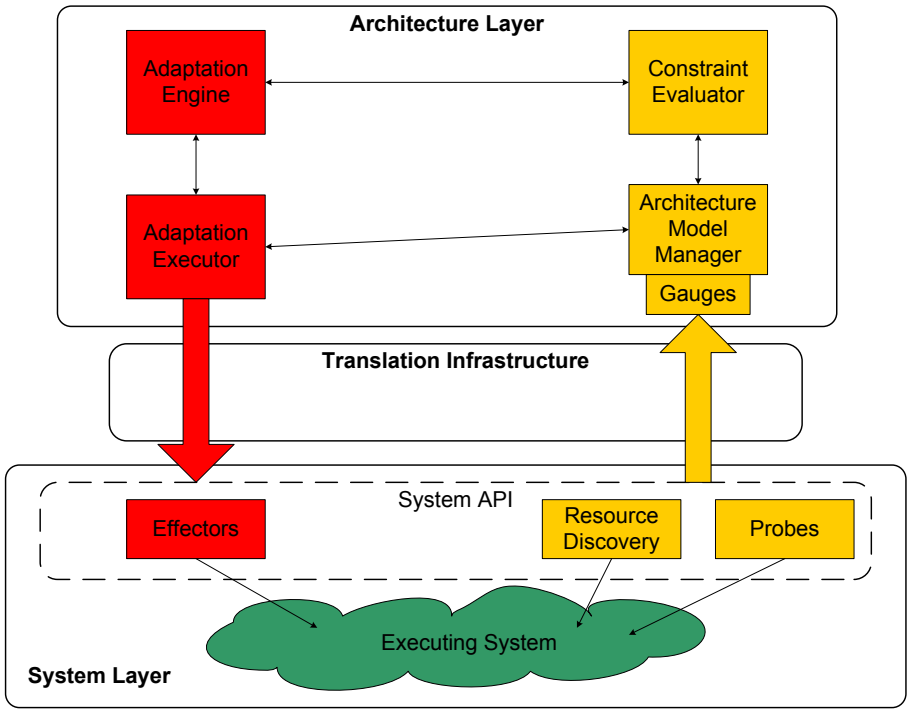


Figure 2.13. Rainbow Framework. (Garlan et al., 2004)

The Rainbow framework associates a software architecture with an executing system. Probes and gauges monitor the executing system. Information received from gauges is abstracted to relate it to the architectural model. The model is a hierarchical graph of components (nodes) and connectors (arcs) annotated with a set of properties, for example, load on a server component. When information received from the monitoring system leads to the value of a property being changed, an analysis is activated. Analysis checks the current properties of the architecture against the architectural style, which is defined as set of types, rules and constraints. On violation of a constraint, repair handling is invoked. A repair strategy is chosen and translated to the running system. Repair strategies are based on the architectural style and must be supported by methods to modify the executing system.

Experimental work (Garlan et al., 2003) has shown that this externalized approach to self adaptation causes significant slowdown and is therefore best suited for global

changes, and acting on long term behaviour trends. The coordination of multiple self-management techniques for a single system has also been considered (Cheng et al., 2004). This requires mediation facilities to decide which repair strategy to choose from a conflicting set.

In Table 2.5, this work is compared against the categories defined in Table 1.1.

Table 2.5. Wright and ACME. This table categorizes the support for evolution in Wright, ACME and Rainbow.

Dynamic Change

Openness	No - The work does not preclude the introduction of new behaviours but concentrates on adaptation rather than the wider problem of evolution.
Maintain State	No
Component Addition and Removal	Yes - Dynamic Wright switches between different configurations in order to add and remove components.
Connections Bound and Unbound	Yes - In Dynamic Wright in the same way as component addition and removal.

Autonomous Change

Automation	Rainbow automates closed adaptation.
Automated Motivations for Change	Probes and gauges provide measurements that are evaluated in terms of the architecture model and constraints. Change is initiated if constraints are violated.
Automated Support for Openness	No
Model Available to Management Components	Rainbow has an explicit architecture model that is kept up to date using information from the monitoring mechanisms.

Change Localization

Higher Order Connectors	Wright introduces connectors as first class entities.
Programmable Connectors	Yes
Partial Decomposition	Higher order connectors can isolate components.

Change Management

Change Management	Rainbow is a change management infrastructure that includes: an architecture model; constraint evaluation; an adaptation engine; and mechanisms to monitor and adapt an executing system.
View of Executing System	Yes - Architecture model is always up to date.
Partial Views	No
Vocabulary for Specifying Changes	Rainbow's adaptation strategies are defined using a set of style specific actions (adaptation operators) and architectural properties (component attributes).
Open Adaptation	No
Closed Adaptation	Rainbow is an external framework for closed adaptation.
Architectural Styles	Yes - Architectural analysis depends on the style constraints and changes are defined in terms of styles.

2.6 ArchStudio and C2

C2 is an architectural style supported by a set of tools (ArchStudio) for the development of robust, continuously available systems (Oreizy et al., 1999; Oreizy and Taylor, 1998; Oreizy et al., 1998). C2 and ArchStudio are designed to be able to *"accommodate unplanned changes and incorporate behaviour unanticipated by the original developers"*. Evolution is facilitated by the C2 architectural style, which specifies explicit connectors, and runtime support for dynamic change in ArchStudio.

C2 is a hierarchical publish-subscribe style comprising a network of concurrent black-box components, which implement the application's functionality and are bound together by asynchronous connectors. Components can only be attached to connectors, but connectors may be attached to other connectors. The topology is defined as horizontal layers of components, with connectors between the layers. Figure 2.14 shows an example C2 architecture, where it can be seen that connectors separate layers of components. Both components and connectors are defined to have a *top* and *bottom* in the architecture. The top of a component may only be connected to the bottom of a connector and visa versa. Connections between two connectors must also be top to bottom. Each component may be connected to at most one connector at the top and one at the bottom. There is no such restriction on connectors.

C2 components can be started, stopped, connected and disconnected. Each component contains a *start* method to initiate its execution and *finish* method to terminate it. The default *finish* method allows a component to complete its current processing before stopping. An implementation of C2 style architecture could also include methods for components to export and import internal state to and from other components.

The use of event-based implicit invocation means C2 is suited to runtime reconfiguration. Event-based implicit invocation is a style where instead of communicating directly, components broadcast events and receive events that they register interest in (Garlan and Shaw, 1996). Components are only aware of services provided at higher levels of the architecture, which they may make *requests* to use. Events, for example a component's internal state change, are broadcast to lower level connectors via *notifications*.

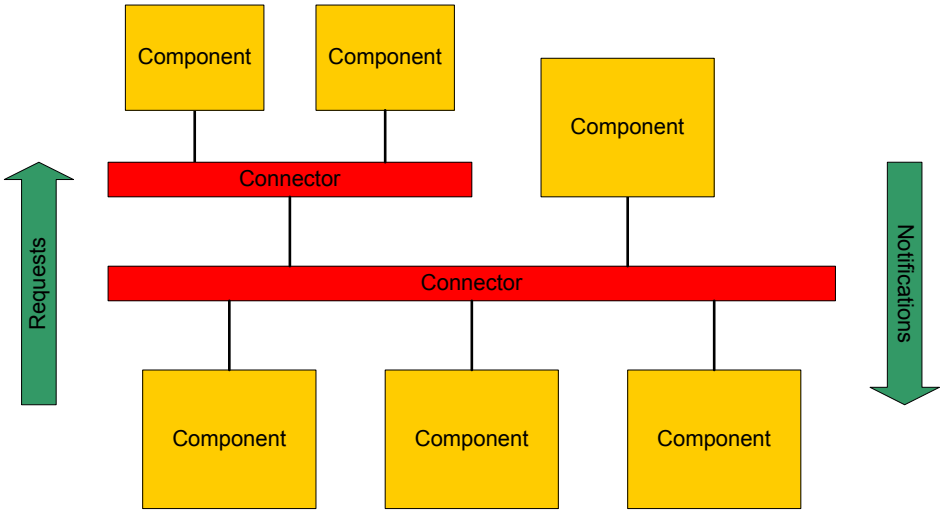


Figure 2.14. C2. Horizontal layers of components and connectors in the C2 style.

In C2, the policy for evolution is encapsulated in the connectors and can be changed without affecting the system's components. Connectors separate component behaviour from interaction, and therefore separate application specific behaviour from decisions regarding *change application policy* and *change scope*. Change application policy controls how change is applied to a running system. During a component update, for example, new requests for services could be directed to the new component and requests for an established service could go to the old component, which would be allowed to finish processing its existing requests before being disconnected. Different connectors may implement different change application policies within the same architecture. Change scope defines the extent to which different parts of a system are affected by a change. During system evolution, connectors control change isolation by holding requests to unavailable components in a queue.

2.6.1 ArchStudio

ArchStudio is based on the Java platform and provides tools to determine what evolutions to apply, reason about the consequences of change and preserve application

integrity. The C2 style is implemented as a class framework where components, connectors and messages are represented by abstract classes. They are therefore explicit entities in the implementation as well as the architecture. There is also an architecture class providing methods for architectural tools to manipulate the architecture, for example, *weld()* and *unweld()* methods connect and disconnect components and connectors. The architecture class subclasses the component class, as components may contain internal architectures.

The interaction of the ArchStudio tools is shown in Figure 2.15 (Oreizy and Taylor, 1998). An architectural model is kept up to date during execution and saved to disk when the system is down. The model takes input from various sources of architectural change. These currently include a graphical tool, an interactive command line interface and modification scripts. The tools can query the architectural model in order to make changes that take the current configuration into account. An architectural evolution manager evaluates the validity of any modifications to the system against a set of constraints. It applies valid changes, specified in terms of styles, to the implementation. It also maintains the architectural model to keep it up-to-date with changes in the implementation. The architectural evolution manager has implicit knowledge of the C2 style rules. Additional constraints are specified using Armani (Monroe, 1998).

A summary of the main benefits of this approach is (Oreizy et al., 1998):

- Evolution at a high level of abstraction eases human understanding.
- There are no restrictions on component internals, allowing off the shelf components to be incorporated.
- Change application policy is separated from application functionality.
- Control of change policy and scope is in the hands of the system architect.

In Table 2.6, C2 and ArchStudio are compared against the categories defined in Table 1.1.

Table 2.6. C2 and ArchStudio. This table categorizes the support for evolution in C2 and ArchStudio.

Dynamic Change

Openness	C2 is designed primarily to support runtime reconfiguration at the component level but ArchStudio accepts change scripts which could introduce new behaviour.
Maintain State	Not currently supported.
Connections Bound and Unbound	Components are connected through explicit connectors and may be connected and disconnected at runtime.
Reflection	Changes are applied using Java dynamic class loading and modification scripts.

Autonomous Change

Automation	ArchStudio supports autonomous change applied by the AEM.
Automated Motivations for Change	Change scripts.
Automated Support for Openness	Change scripts are a mechanisms for automatically introducing new behaviours but not for defining them.
Model Available to Management Components	AEM maintains an up-to-date model.

Change Localization

Higher Order Connectors	C2 Style requires higher order connectors.
Programmable Connectors	Yes
Partial Decomposition	Change localization can be achieved by disconnecting a component and therefore isolating it.

Change Management

Change Management	ArchStudio supports a managed change process including architectural analysis on an explicit model.
View of Executing System	ArchStudio contains an explicit Architectural Model
Partial Views	No
Vocabulary for Specifying Changes	Changes specified by methods in the implementation interface.
Open Adaptation	New behaviours can be introduced through change scripts and user interaction.
Closed Adaptation	No
Architectural Styles	C2 is a specific style.

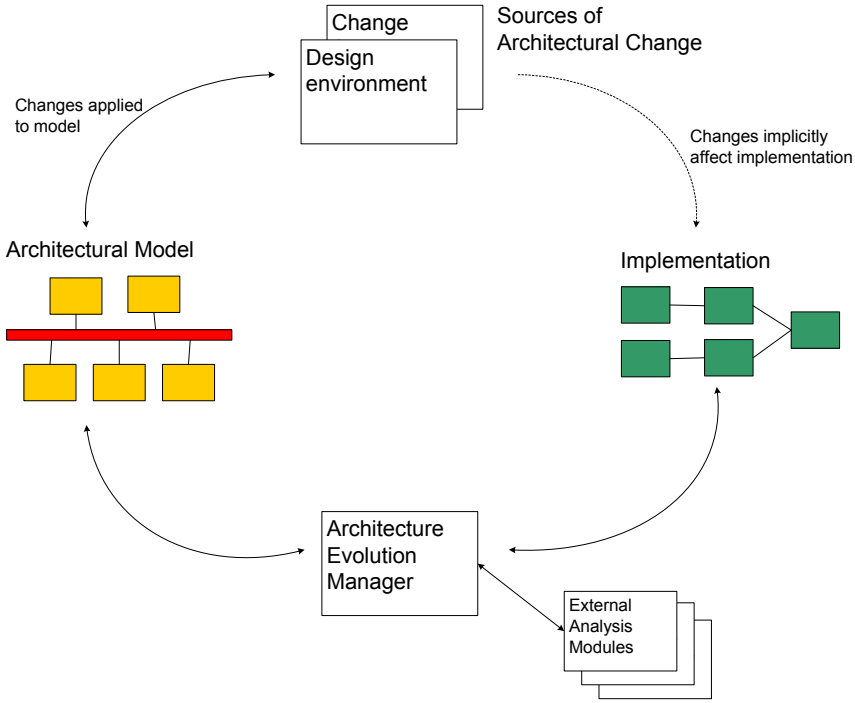


Figure 2.15. ArchStudio. High level interaction of ArchStudio tools (Oreizy and Taylor, 1998).

2.7 Intentional Programming

Intentional Programming (IP) is an Integrated Development Environment (IDE) where program data is incorporated into the source code (Czarnecki and Eisenecker, 2000; van Wyk et al., 2001; Simonyi, 1996; Simonyi, 1995). In order to incorporate data, programs under development are modelled as syntax graphs, which include links to data values stored outside the program, e.g., in the file system. The links are abstracted over in the user interface. For example, data values from a database may be displayed diagrammatically. The IP environment uses meta-programming techniques to convert program text to the graph model and then display it to the user as the program is being written. Meta-programming is also used at compile time to generate compilable code from the graph.

The IP programming environment is supposed to support a style of programming based on the developer's *intentions*. Intentions express programming abstractions as language features. They aim to avoid both loss of information and unnecessary programming language clutter. Information that may otherwise be lost, when a program is represented only as source code, includes documentation, as well as program data stored in databases or files that may be moved. Unnecessary clutter, for example, complicated syntax to define a matrix that could instead be displayed graphically, is abstracted out by the input and display methods. The IP environment is developed using meta-programming and can be evolved, from within the environment, through the definition of new intentions and alteration of existing intentions. The IDE is therefore customizable and extensible. Because it is designed for the introduction of new intention abstractions, the IP environment is capable of open evolution.

The IP representation is based on an *active source graph* characterizing the program under development. The graph is described as *active* because it is updated as the program is written. Figure 2.16 shows the elements of the active source graph and the transformations between them. The programmer interacts with a visual rendering of the source code. As the programmer *edits* the code, a source graph is built up and altered, prompting updates to the visualization. On compilation, the source graph is *transformed* into a reduced code that can be processed by a compiler. Transformations may include the addition of *debugging* information in the code, which will be passed back to the programmer through the source graph.

The source graph is an abstract syntax tree augmented by edges between nodes and their declarations, making it into a graph. An example of an *if* statement represented

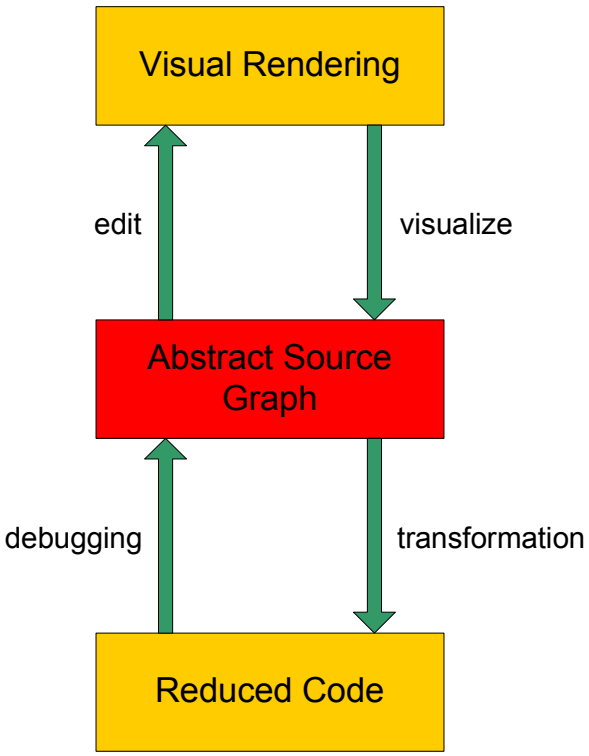


Figure 2.16. Intentional Programming. The IP development process.

by a source graph is illustrated in Figure 2.17. In the upper left hand corner is a node representing *if*. It is parameterized by: a *condition* node, a *then* node and an *else* node. Parameterization is represented by an edge in the source graph. The *if* node is an *intention instance* and contains a reference, another edge in the graph, to its *intention declaration*. The *intention declaration* defines methods associated with an *if* statement and itself references a declaration node defining methods associated with declarations. Every node in the source graph is an instance of an abstraction, and every abstraction has a corresponding declaration. Some declarations are built into the language, such as the *if* statement. Others are created as program variables are defined.

Intention declarations are associated with a number of methods defining the behaviour of the intention. They include, but are not restricted to, visualization, browsing, compilation, debugging and versioning. These methods are called at programming time. For example, the visualization method is called to display an instance of an intention:

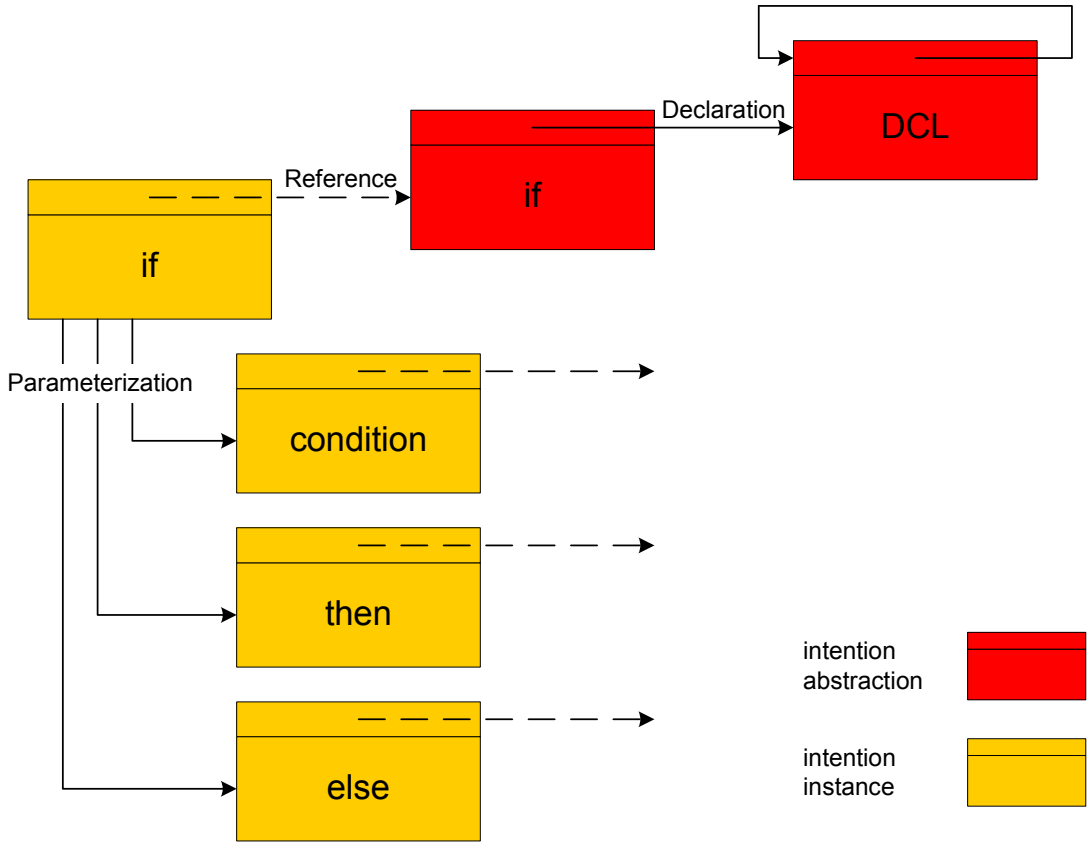


Figure 2.17. Intentional Programming. The IP syntax tree.

in the case of a mathematical formula it might provide a graphical display. The compilation method for the same formula might generate optimized C code. It is expected that developers would construct their own domain specific intentions to tailor their programming environment.

The foundation of the IP system is a reduction engine that produces reduced code from the active source tree. The reduced code is a general set of low-level abstractions, such as would be contained in a simple programming language. It is a necessary step for intentions that are not expressed in a programming language.

Because the information about compilation and parsing in IP is localized, the representation permits programs constructed from multiple programming languages. Different programming languages operating together must be mapped to the reduced code before being compiled. Many abstract syntaxes may be mapped to the single concrete syntax of the reduced code. Object code, e.g., Java bytecode, is then generated from the reduced code.

There are a number of similarities between the MPF and IP:

- Hypercode graphs and active source graphs both use a program representation that includes program syntax and data .
- Both integrate changes using generative programming.
- Both systems are developed in themselves, allowing their respective frameworks to be evolved.

Rather than being the same however, the systems are complementary, because IP concentrates on providing an environment for software at development time, whereas the MPF focuses on an environment for evolving software at runtime.

In Table 2.7, IP is compared against the categories defined in Table 1.1.

Table 2.7. Intentional Programming. This table categorizes the support for evolution in Intentional Programming.

Dynamic Change

Openness	Yes - New intentions can be defined from within the environment.
Maintain State	No
Connections Bound and Unbound	Not applicable as IP is not modelled using software architectures. However, new intentions can be added and existing intentions updated.
Reflection	New intentions are compiled into Dynamically Linked Libraries (DLLs) that are can be imported into the IP environment.

Autonomous Change

Automation	No
Automated Motivations for Change	No
Automated Support for Openness	No
Model Available to Management Components	No

Change Localization

Higher Order Connectors	NA
Programmable Connectors	NA
Partial Decomposition	NA

Change Management

Change Management	IP is designed to incorporate changes in the form of new intentions and changes to existing intentions. However, there is no management for the change process.
View of Executing System	No
Partial Views	No
Vocabulary for Specifying Changes	New intentions are defined using C programs that are compiled into DLLs.
Open Adaptation	Yes - The introduction of new intentions.
Closed Adaptation	Existing intentions can be altered through developer interaction.
Architectural Styles	No

2.8 Summary

In this chapter, a number of ADLs and frameworks for evolution based on software architectures are described.

Darwin An ADL that supports change through lazy and direct dynamic instantiation. The former instantiates and connects predefined component types dynamically. The latter allows newly defined component types to be instantiated in an environment of predefined connections. Self-organizing systems based on Darwin use constraints to motivate change.

Gerel An ADL for generic reconfiguration, changes in Gerel can be pre-programmed to operate over a generic structure. Generic changes can remain valid as the components and topology in the system evolve.

Weaves An ADL that separates system observation from data capture, Weaves paved the way for probes and gauges, which are now used by most systems that implement a closed adaptive engine.

ArchJava ArchJava is an extension to Java to include software architecture constructs, such as components and ports in program source code. The implication of integrating the software architecture and source code is that ArchJava systems can use a compiler to enforce architectural constraints in the implementation.

Wright, ACME and Rainbow An interchange language for ADLs, ACME is based on Wright and has been used in a framework for evolution. The Rainbow framework includes components to establish when, why and what evolution should take place according to a set of constraints. Evolutions are applied to an executing system that provides feedback to the management components using probes and gauges.

ArchStudio and C2 ArchStudio is a framework to support evolution in systems built in the C2 style, a hierarchical, publish-subscribe style that accommodates change. An Architecture Evolution Manager compares the current state of a system against a set of constraints. If constraints are violated, an Architectural Model and the executing system are both adapted by the Evolution Manager, ensuring that the Model is always in sync with the system.

Intentional Programming An environment for program development, IP differs from the other systems described here in that it does not approach evolution from an architectural perspective. It is, however, an example of a system using a program representation that includes both source code and data. IP uses meta-programming to manage the transformation from program representation to executable code.

2.9 Conclusion

ADLs are used to model software architectures and are suitable for understanding and analyzing systems. The ADLs discussed in this chapter have an extra dimension in that they model various aspects of evolving systems. However, because ADLs consider software architectures separately from their implementations, they do not provide mechanisms for applying the changes they model to executing systems. ArchJava is an exception in that it integrates software architecture and source code, but it does not address the process of evolving the system. The MPF provides the mechanisms to effect the evolutionary changes that are modelled by ADLs.

The evolution frameworks explore the process of evolution and include mechanisms for change management as well as tools to apply changes. Rainbow (Sec. 2.5) is an external framework for adaptation that relies on effector mechanisms to carry out the adaptations. These could be supplied by the MPF, however that would require a system designed for evolution and is therefore incompatible with the notion of external adaptation. Changes applied by ArchStudio (Sec. 2.6) are restricted to the changes allowed by a predefined interface. The MPF provides an interface to the executing system that does not restrict the changes that can be applied. Instead, restrictions can be layered on top of the MPF with respect to the policy of a particular change management framework. It also makes available a representation of any value in the executing system that is always complete and up-to-date and can be used for introspection and analysis.

Chapter 3

Orthogonal Persistence, Structural Reflection and Hypercode

This chapter explains the set of technologies on which the Meta-Programming Framework is based. These include: persistence, structural reflection, Hypercode and the ArchWare framework.

3.1 MPF Technologies

The Meta-Programming Framework's support for a software architecture approach to evolution requires a set of base technologies that support an incremental evolution process. These technologies have been developed by other researchers and the MPF builds on their work.

- Orthogonal persistence provides referential integrity.
- Structural reflection creates new values during execution and allows introspection.
- Hypercode gives a representation of an executing program.
- An ADL is used to model software architectures and partially decompose executing systems.

The referential integrity offered by orthogonal persistence guarantees that once a reference has been established, the value that it references will remain accessible for the lifetime of the reference. This prevents errors occurring because of dangling references to, for example, data stored in files or databases which has been moved or deleted. Referential integrity is particularly useful in long-lived, complex, evolving systems where it may be difficult to establish the set of values that contain references to some data, i.e., the set of values which would have to be updated if the data were moved. An orthogonally persistent platform provides referential integrity for the MPF.

The creation of new values during execution and introspection are made possible by structural reflection. Support for the creation of new values during execution is a prerequisite for evolving an executing open system. In the MPF, introspection allows a meta-program to access a component's internal state. The information may be used to determine whether or not the component needs to be evolved and the state may be retained over a component update.

Hypercode is a program representation capable of characterizing executing programs. This implies that Hypercode, in contrast to plain source code, is a complete representation that incorporates a program's state and data. This is integral to the MPF where a meta-program can evolve a system by operations on a Hypercode graph, a program representation including both code and data.

There are many ADLs available to model software architectures but the MPF requires language support for evolutionary mechanisms. The ArchWare ADL, designed as part of a framework for evolving software systems, is used as an experimental base. The primary evolutionary mechanism distinguishing ArchWare ADL is the decompose operator, which is used to partially stop a system so that it can be incrementally evolved while the rest of the system continues to execute uninterrupted.

This chapter explains concepts fundamental to an understanding of the environment in which the MPF has been developed. These are orthogonal persistence, structural reflection, Hypercode, and the ADL and its framework. The properties of ProcessBase, a programming language incorporating both orthogonal persistence and structural reflection which has been used to implement the MPF, are also described. The section on Hypercode starts with an explanation of Hyper-Programming from which Hypercode was developed. The chapter finishes with an explanation of a Hypercode implementation, for use in the MPF, that was constructed as part of the work for this thesis. It is reusable for different programming languages and provides an interface to the Hypercode operations.

3.2 Orthogonal Persistence

Orthogonal persistence (Morrison et al., 1996; Atkinson and Morrison, 1995; Atkinson et al., 1983) removes the unnecessary distinction between short and long term data. The manner in which data is manipulated is independent of its persistence and the same mechanisms operate on both short and long term data. Orthogonal persistence implies that the lifespan of data is unrelated to its type or the way in which it was created. In conventional programming environments, data storage mechanisms are dependent on the lifetime of the data. For example, long term data is written out to the file system or a database. Access to the data is determined by the storage mechanism.

Persistent systems are designed according to the following principles (Atkinson et al., 1983):

The Principle of Persistence Independence The form of a program is independent of the longevity of the data it manipulates. Programs look the same whether they manipulate short-term or long-term data.

The Principle of Data Type Orthogonality All data objects should be allowed the full range of persistence irrespective of their type. There are no special cases where objects are not allowed to be long-lived or are not allowed to be transient.

The Principle of Persistence Identification The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system. The mechanism for identifying persistent objects is not related to the type system.

The principle benefits of persistence are simpler semantics and referential integrity (Atkinson et al., 1983; Atkinson and Morrison, 1995). Improved programming productivity follows from simpler semantics, because in a persistent environment a programmer does not need to write code to transfer data between short and long term storage mechanisms. Without persistence ad hoc arrangements for long term data storage and data translations are necessary. In addition to improving productivity, persistence can improve accuracy, because type checking protection mechanisms operate over the whole environment. Strong typing mechanisms mean that the type system extends over the lifetime of the data, preventing mistakes that occur when, for example, data is input from the file systems and cast into the wrong type. Referential integrity is automatically supported in a persistent environment. This means that once a reference to an object has been established that object will remain accessible for the lifetime of the reference. Type correctness is also implied. Referential integrity is one of the requirements for Hypercode (Sec. 3.5).

Orthogonal persistence provides a programming environment which supports incremental evolution in long-lived systems. All computation takes place within a persistent environment and it is therefore possible to evolve the system while it is executing. Programming support mechanisms have been built on top of persistent systems to take advantage of the benefits provided by persistence. These include programming languages and environments, such as ProcessBase and Hypercode, as well as structural reflection.

3.2.1 Existing Persistent Systems

A number of persistent programming systems have been implemented. PS-algol (Atkinson et al., 1982), which extends S-algol, and Napier88 (Morrison et al., 2000b) were prototype persistent programming languages. PS-algol and Napier88 have:

- A persistent store, in which all values are reachable from a root object.
- An infinite union type (Morrison et al., 1990), supporting dynamic injection and projection operations. An injection operation takes values of type T and returns values of type C(T), where C is some type constructor. Projection is the inverse of injection.
- First class procedures (Atkinson and Morrison, 1985), which can be placed in the persistent store and therefore access any other data value in the store.

ProcessBase (Morrison et al., 1999a) was developed based on the work done for PS-algol and Napier88. It is more fully described in Section 3.4.

Other persistent languages include: DBPL (Schmidt and Matthes, 1994), which uses modules to encourage good software engineering practices; O2 (Deux, 1990); Galileo (Albano et al., 1985); E (Richardson and Carey, 1989), an OO persistent language based on C++; and PJama (Atkinson et al., 1996a; Atkinson et al., 1996b), orthogonally persistent Java. The Esprit project on Fully Integrated Data Environments resulted in a body of published work on persistent systems (Atkinson and Welland, 2000). Many other languages have been extended to include persistence. Examples include Smalltalk (Straw et al., 1989), and C and C++ (Hosking and Novianto, 1997).

The PJama Persistent Platform (Dmitriev, 2000) builds support for evolution on top of PJama. It concentrates on increasing the safety of development time evolution by preventing propagation of incompatible changes into the store. The system uses persistent build technology to do smart compilations, where only changed classes are recompiled. It can also perform automatic object conversion on instances whose class definitions have been changed.

3.3 Structural Reflection

Structural reflection provides the ability to incorporate new program fragments into the ongoing computation (Stemple et al., 2000; Kirby, 1992; Dearle, 1987). It is therefore an essential tool in the development of systems which can be dynamically evolved during execution. In a persistent environment, structural reflection supports type-safe evolution of long-lived programs and data, and the specification of highly generic programs that may be re-used in many contexts. In strongly typed systems the reflection process includes checking of the generated program fragments to ensure type safety.

To support structural reflection a system requires (Morrison et al., 1996):

- A callable compiler.
- First class code.
- An infinite union type with injection and projection.

A callable compiler is necessary for structural reflection in strongly typed languages. It is available as a function which takes a string parameter containing the code to be compiled at runtime. The compilation produces some object code, which must be bound into the executing program to include it in the current computation. This is where a *coerceToValue* function is needed to bind and type the result of compilation. It allows the result of compilation to be introduced into the system as an executable function, i.e., it casts the object code as a language type ensuring that the compiler produces a value that can be used by the enclosing program in a type-safe system.

In a language with first class code, functions (or methods, or procedures) can be treated in the same way as other language values. For example, they can be assigned to variables and passed as parameters. Structural reflection requires first class code in order to introduce the result of compilation into the executing system.

In a strongly typed language, the callable compiler's return type is defined in advance. However, the type of the program it is compiling is determined dynamically. Therefore, the result of compilation is an infinite union type. The dynamic projection operation is used to cast the value of infinite union type onto its specific type. At this point a new program has been compiled and bound and the structural reflection is complete.

3.3.1 Definition of the Reflection Operation

The reflection operation is defined more rigorously in terms of mappings between domains (Stemple et al., 2000). The language, or representation, domain R contains expressions in the programming language. The domain of values, or entities, E contains data and executable programs. Reflection maps values from the R domain to the E domain:

$$\text{Reflection} : \text{R} \rightarrow \text{E}$$

The process takes place in three stages:

1. Generate a program representation within *R*. For example, generate a string consisting of valid program statements.
2. Transform the program representation in *R* into an executable value in *E*. For example, use a callable compiler to compile the generated string.
3. Bind the executable into currently executing system. For example, use the *coerceToValue* function to introduce the result of compilation into the current environment.

Generation, the first step, involves producing a string value containing source code from the *R* domain. The generated code may depend on current program state or conform to a static template. The second step is the task of the callable compiler which converts a string from the *R* domain into object code. This is a mapping from *R* to another language domain *R'*. In the third step, *coerceToValue* maps the value from *R'* to *E*. The progression can be defined:

$$\text{Reflection} : \text{R} \rightarrow \text{R}' \rightarrow \text{E}$$

The functions involved act in the following order:

$$\text{bind (compile (code))}$$

The *code* is *compiled* by the callable compiler. Then the *coerceToValue* function is used to *bind* it into the currently executing system, producing a new value. An intuitive description of the process states that: *during the evaluation of a reflective procedure the result of the evaluation is itself evaluated as an expression in the language* (Stemple et al., 2000).

3.3.2 Reification

Reification transforms an abstract entity into a concrete representation. In terms of the *R* and *E* domains it is the opposite of reflection:

$$\text{Reification} : \text{E} \rightarrow \text{R}$$

Reflection is often used to create new values that depend on current values in the computation. In the code generation step, these values must be represented in the language domain. Reification can be used to transform the values in the currently executing domain into a representation in the language domain. As an example, consider the reification of the boolean value *true*, which could be represented as the string `'true'`.

As well as program and data values in the *E* domain, types may also be reified. A *typeOf* function provides access to the types of the changing system. In programming languages with an infinite union type, reification on the type system allows the specific type of values to be discovered dynamically. For instance, a new function created using reflection may have a statically unknown return type. The compiler initially casts it as the infinite union type and reification of the type system is then used to find out what type it actually returns. Code can then be generated to cast the new function onto the correct type.

3.3.3 Implementations of Structural Reflection

Languages which provide runtime structural reflection include:

- Lisp (Anderson et al., 1986) and POP-2 (Burstall et al., 1971), which have untyped, runtime structural reflection.
- PS-algol (Atkinson et al., 1982), Napier88 (Morrison et al., 2000b) and ProcessBase (Morrison et al., 1999a) are languages developed for a persistent environment with strongly typed, runtime structural reflection.
- CRML (Hook and Sheard, 1993) and its successor MetaML (Sheard, 1998) are strongly typed, compile time reflective languages.
- Java has been augmented with strongly typed, runtime reflection (Kirby et al., 1998).

3.4 ProcessBase

The ProcessBase Language (Morrison et al., 1999a) was developed collaboratively to combine persistence with the language requirements for process modelling (Warboys et al., 1999b). The language design is guided by three semantic principles (Strachey, 1967; Tennent, 1977):

The principle of correspondence states that the rules governing the use of names and bindings in a programming language should be consistent.

The principle of abstraction states that for all significant syntactic categories in the language there should be an abstraction mechanism.

The principle of data type completeness states that any combination or construction of data should be allowed in all types, resulting in all data objects in a language being first class.

A ProcessBase system entails the ProcessBase language and its persistent environment. The model of persistence is reachability from a root object. Types are sets of values from the value space, including recursive types. They are mostly statically checkable and exhibit structural equivalence.

The tools necessary for structural reflection are part of the ProcessBase language. A callable compiler is implemented in ProcessBase. It is passed the code to be compiled as a parameter of type `string` and returns the result of compilation as well as a data structure describing the type definition of the compiled code. It is provided in the form of a `compile` function. Consider the following example:

```
let newValue ← compile('1 + 2')
```

The `compile` function is passed a string of ProcessBase code and the result of compilation is stored in `newValue`. *coerceToValue*, the function that introduces new values into the current environment, is part of the callable compiler. `newValue` is typed as *any*, the infinite union type in ProcessBase. The callable compiler returns type *any* because the type of the code is not known in advance. If the code does not return a value it will be typed as a function. Otherwise, the type will be a function that returns an *any*.

The dynamic projection operation is used on `newValue` to cast it as a function type. ProcessBase has first class code, meaning that functions can be passed as parameters and returned as the result of other functions.

```
1  project newValue as X onto:
2  function → any: {
3      let result ← X()
4      project result as Y onto:
5      int: Y is 3
```

```
6       default: Error
7   }
8   default: Compilation Error
```

After the callable compiler returns, the result (`newValue`) is projected onto the type of a function that returns an `any` (line 2). If the projection fails, meaning a compilation error has occurred, the `default` clause is executed (line 8). In this example, the compiled code returns an integer (`1 + 2 = 3`). The code is executed (line 3) and the result projected onto an integer type (line 5).

Reification of the type system is made possible in ProcessBase by the *typerep* function. It takes a value of the infinite union type and returns a representation of the value's specific type. *typerep* is used to discover the statically unknown type of new values created by structural reflection.

The properties of ProcessBase which support the MPF include:

- Structural reflection, which allows meta-programs to update other programs without stopping the execution.
- Reification of the type system, which allows new values of statically unknown type to be introduced into a strongly typed environment.
- A persistent store, which permits meta-programs to operate over persistent data and provides referential integrity.

3.5 Hyper-Programming

Work on persistence, together with structural reflection, led to the idea of Hyper-Programming (Kirby et al., 1992). In persistent languages with first class code, the persistent store contains functions (or methods or procedures) that reference data and other function values. A representation of their structure gives developers some understanding of the contents of the store and helps in writing new programs that use values in the store. Provision of such a representation requires a model including both data and function values. This is the basis for Hyper-Programming and Hypercode, which advocate a single view of the computation space.

A hyper-program contains links embedded in the source code to data in the persistent store. Hyper-programming allows the programmer to write less code, since a

textual description of how to access data is replaced by a link to the data. It also leads to more reliable code because type checking of linked items can be performed statically rather than dynamically. The first hyper-programming system, implemented for Napier88 (Kirby et al., 1992), demonstrated how the technique could ease the task of reflective programming and provide support for source representations of procedure closures.

Hyper-programs are part of the persistent environment. Because persistent data values are available during program composition, direct links exist from a hyper-program to the values it references. In addition, the hyper-program source is resident in the persistent store. By this mechanism, a current version of a value's hyper-program source code is always available to the programmer. Reflection is used to support the conversion of hyper-programs into executable programs. New sections are compiled in isolation from the original source program and the results linked back into the executing program.

In the implementation of hyper-programming for Napier88 (Kirby et al., 1993) the user views values from the persistent store through a browser. The browser provides a visualization of the objects in the persistent store. A hyper-program is composed by writing source code and copying hyperlinks from the store into the program.

3.6 Hypercode

Hypercode (Zirintsis, 2000) builds on the concepts introduced by hyper-programming to abstract over the differences between source code, executable code and data for the programmer. Thus the task of the programmer is made easier, as they are (Zirintsis et al., 2001):

> *presented with a simpler environment in which the conceptually unnecessary distinction between these forms is removed. In terms of Brooks' essences and accidents (Brooks, 1987), this distinction is an accident resulting from inadequacies in existing programming tools; it is not essential to the construction and understanding of software systems.*

In a Hypercode system the user composes Hypercode and the system executes it. The user only sees a single view of the system and underlying operations are abstracted over.

As in hyper-programming, a Hypercode program is constructed from a mixture of text and hyperlinks. The text is normal program source code and the hyperlinks point to existing values. In a Hypercode system the user can compose programs interactively, navigating the environment and selecting data items, including functions, to be incorporated into their program as hyperlinks (Kirby et al., 1992). Clicking on a hyperlink allows the user to see a Hypercode representation of the value. The artificial distinction between source and executables is removed, therefore a Hypercode view can be generated for any value in the system (Morrison et al., 1999b).

Hypercode requires referential integrity and structural reflection. Referential integrity is necessary to ensure that representations including hyperlinks remain valid for their lifetime. It can be provided by a persistent system. Structural reflection is used to implement the introspective tools that allow Hypercode representations to be available throughout the software life cycle. It is also necessary for the evaluation of Hypercode programs.

3.6.1 Properties of Hypercode

Hypercode exhibits a number of properties that aid evolution in long-lived dynamic systems. Hypercode representations characterize program closure at any stage of the software life cycle, including execution. They therefore provide a programmer with a comprehensive view of the program at any time, including the current state of internal variables. Using a Hypercode model in the MPF allows the framework to provide an interface that exposes program closure to meta-programs.

Sharing is modelled by permitting any number of links to the same value. In Figure 3.1, which shows some Hypercode with three hyperlinks pointing to values, two separate hyperlinks are pointing to the same value, which may be an executable or data value.

Another feature of Hypercode which underpins its usefulness in the evolutionary context is its ability to preserve the state and shared data of a system during evolution. This transpires because the Hypercode source, a mixture of text and links to values in the store, is always available. When components in the system are changed or updated, hyperlinks can be maintained since they exist in the Hypercode representation, (in the representation domain), as well as in the executable, (in the entity domain). Underlying persistence guarantees referential integrity, which implies that the hyperlinks will

always be accessible, therefore internal state and shared data can be preserved over changes in a dynamic system.

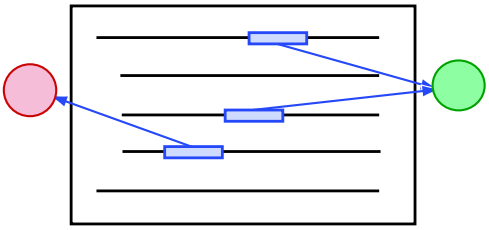


Figure 3.1. Sharing. Hypercode permits any number of links to the same value.

Hypercode supports introspection and the Hypercode view of a value is always available. This eases the task of writing evolving processes, because the user can see a concrete representation of the results of decomposition. Evolving processes can otherwise be hard to write and understand because of the problem of dealing with extant data, particularly if the required evolution depends on the current state of the process being evolved (Greenwood et al., 2003). When introspection is combined with compilation and binding the technique known as structural reflection results (Kirby et al., 1998). Meta-programs can use introspection to determine the current state of program values and retain it over changes. For example, probes could be implemented that use introspection to gather information.

3.6.2 Entity and Representation Domains

Program and data *values* and their *representations* are different and serve distinct purposes but are related because every value has an equivalent representation. Considering them to reside in two separate domains (Zirintsis, 2000), the *entity* domain for values and the *representation* domain, aids understanding of their roles and interactions. The entity domain contains all the first class values defined by the programming language. The representation domain contains concrete representations of the values in the entity domain. In the Hypercode context, values exist and execute in the entity domain, but the user views the values in the representation domain.

Figure 3.2 shows the two domains. Values in the entity domain have corresponding views in the representation domain. The links between values in the entity domain are shown as hyperlinks in the representation domain.

Switching between the two domains is achieved with the domain operations *reflection* and *reification*. Reflection places Figure 3.2 shows the two domains. Values in the entity domain have corresponding views in the representation domain. The links between values in the entity domain are shown as hyperlinks in the representation domain.

Switching between the two domains is achieved with the domain operations *reflection* and *reification*. Reflection produces an entity equivalent to a Hypercode representation. In practice, this means compiling some Hypercode source to produce an executable or data value. Reification gives a user viewable representation of an entity in the representation domain. For example, the reification operation is used to produce a view for the user of a function that is executing. The internal domain operations are *execute*, in the entity domain, and *transform*, in the representation domain.

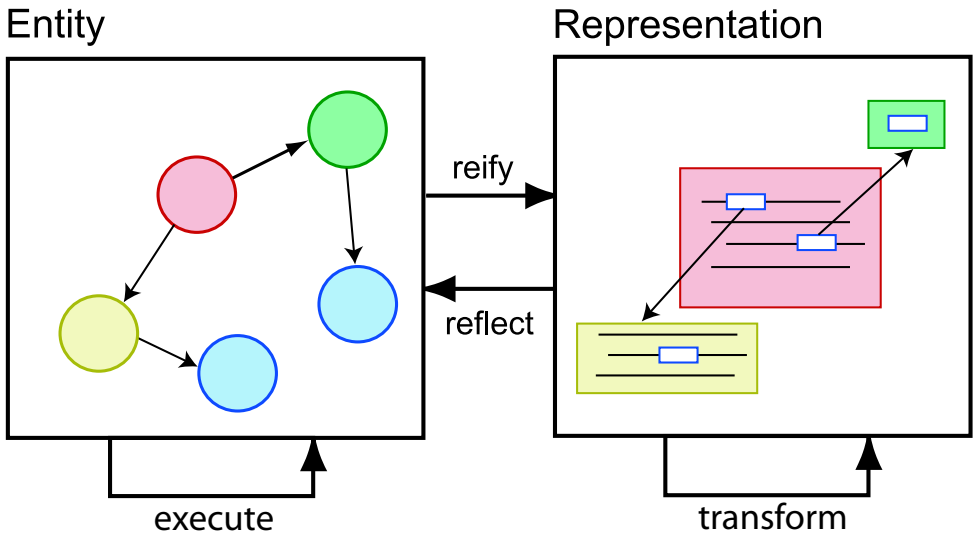


Figure 3.2. Entity & Representation Domains. Hypercode values exist in the entity domain. The user views them in the representation domain.

3.6.3 Hypercode Operations

Hypercode systems have been constructed for Java and ProcessBase (Zirintsis et al., 2001). Both systems implement a set of *Hypercode operations*, through which a user interacts with Hypercode. The Hypercode operations are described in Figure 3.3 (Zirintsis, 2000).

Figure 3.4 is an example, in ProcessBase, of the user view of the explode operation. A function *processor* is defined, inside which two calls to the *process* function are made,

evaluate Compile, link and execute some Hypercode.

explode Give the programmer a Hypercode view of a value pointed to by a hyperlink.

edit Conventional editing facilities.

implode Hide the view of an exploded value.

Figure 3.3. Hypercode Operations. The Hypercode operations define user interaction in a Hypercode programming environment.

with different input data. Inside *processor* are three hyperlinks, two of which point to the same value: the *process* function. Clicking on a hyperlink to *process* reveals the source code of the function. Clicking on a hyperlink to *input1* reveals the string representation of the value. There is no differentiation between the user views of the source code and data value. The exploded hyperlinks can be imploded by closing the window displaying the exploded value.

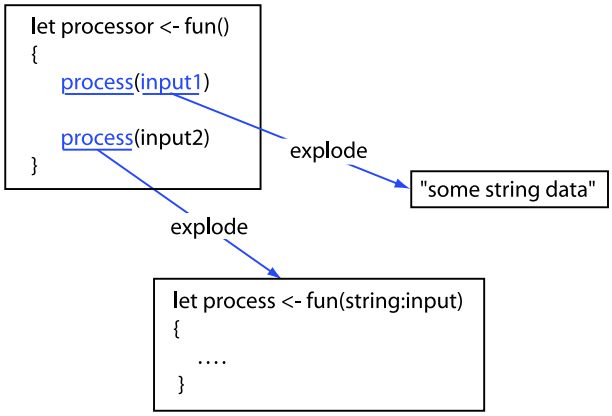


Figure 3.4. Explode. Clicking on a hyperlink explodes it, returning a Hypercode representation of the value.

The Hypercode operations are defined in terms of the entity and representation domains:

evaluate Reflect a Hypercode representation into the entity domain where it executes. Then the execution and its result are reified to produce visualizations in the representation domain. If the execution does not return a result the reification is not performed.

explode Reify an entity to give its representation.

edit Transform a representation. Editing takes place entirely in the representation domain.

implode A subset of the editing facilities.

3.6.4 Octopus

Octopus (Farkas and Dearle, 1994) is a system that extends hyper-programming in that it operates on a program representation including both source code and program values and provides an API to object closure. A data graph is used to represent the closure, where each node in the graph is an object containing a set of references to other objects in its own closure. Octopus's interface allows programs to traverse this graph and manipulate it by adding and removing bindings, therefore evolving the graph. A source code representation of objects is also available in Octopus. However, source code does not contain direct links to the data graph. For example, the source code of a procedure is available as text and the closure of that procedure is available as a separate graph. The data graph in the MPF is similar to that in Octopus, but it is integrated with a representation of the program syntax.

Octopus and the MPF use similar techniques to compile code that includes values from the data graph. The code is wrapped in a function that can take the values as parameters after compilation. Therefore, existing values can be bound into a new function without changing the compiler. The MPF's method is further explained in Section 7.3.1.

In addition to the work on the data graph representation, Octopus provides a common abstraction over all values, allowing values to be inspected or manipulated in a type independent manner. The Octopus interface comprises a set of operations which can be used to provide higher level tools based on the structural reflection in the language[1].

Another aspect of the work is *partially resolved hyper-programming*, which enables the production of templates. The templates allow programs to be constructed and compiled without the requirement that the values used by the program be present. In this manner, individual components can be constructed independently and later assembled to form a complete application.

[1]Thanks to Alex Farkas for his comments.

3.7 ArchWare

The ArchWare project (Morrison et al., 2004; Balasubramaniam et al., 2004b; Balasubramaniam et al., 2004a; Morrison et al., 2003; Greenwood et al., 2003) is concerned with design and construction of an environment for the development of evolvable software. The Meta-Programming Framework presented in this thesis has been applied in the ArchWare Framework.

ArchWare aims to develop Active Architectures, which are defined to be:

- Dynamic in that the structure and cardinality of the components and interactions are not statically known.
- Updateable in that components can be replaced dynamically.
- Evolvable in that the executing specification may be changed.

Elements of the ArchWare environment include

- An architecture description language: ArchWare ADL.
- Languages for analysis and refinement of software architectures.
- A suite of tools based on the ArchWare ADL for design and analysis.
- A runtime framework.
- Process models for evolutionary architecture-centric development.

Evolutionary changes can be expressed differently for different levels of abstraction. Ideally, software systems can be evolved at every level. In ArchWare, software systems are divided into three levels of diminishing abstraction, each with its own evolutionary processes and mechanisms. Figure 3.5 illustrates the different levels: architectural styles, software architectures and the ArchWare Virtual Machine (VM). Representations at higher levels of abstraction can be extracted from the lower levels. These same representations can be instantiated to create systems at lower levels.

At the most abstract level are architectural styles. A style represents a class of systems and can be evolved by changing the constraints on topology and behaviour. Instantiating a style generates a concrete software architecture, whose structure and behaviour

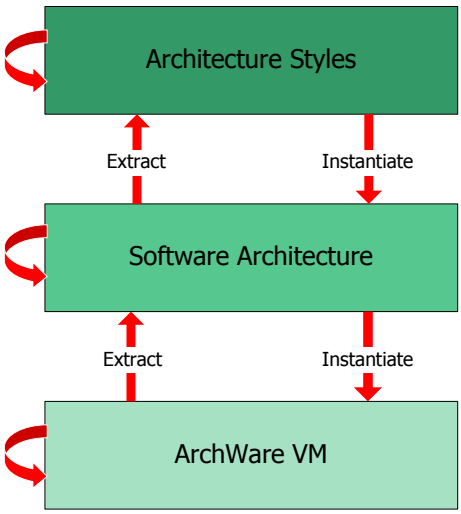


Figure 3.5. Evolution abstractions. Evolution takes place at three levels of abstraction, each with its own evolutionary mechanisms and processes.

conform to the style. The architecture can be evolved through the set of changes to components and connectors allowed within the style. The mechanisms for evolution at the software architecture level are:

- Hypercode, which provides introspection and a program representation.
- Structural reflection, which supports the introduction of new program values at runtime
- A set of language mechanisms available in the ArchWare ADL (Sec. 3.7.4).

The ArchWare VM is at the lowest level of abstraction. Concrete architectures from the software architecture level are compiled to produce a set of executing *behaviours* for the VM. Behaviours contain application functionality and are the ArchWare ADL incarnation of components. Mutable locations and assignment are the mechanisms for change at this level. Software architecture information can be extracted from the behaviours running on the VM because Hypercode provides introspection on the executing system. Architectural style information is contained in software architecture definitions and can be extracted.

Software architectures developed using the ArchWare methodology are:

- **Dynamic** The structure and cardinality of the components may be changed at runtime.

- **Updateable** Components can be replaced.
- **Decomposable** The executing system may be partially stopped and split into its constituent components.
- **Reflective** Specifications of components and their interactions may be evolved at runtime.

Evolution in ArchWare is supported by the *compose* and *decompose* operations of the ArchWare ADL (Sec. 5.3.1). Compose gives a handle to a set of communicating behaviours. Decompose breaks connections between the composed behaviours, which then execute to their reduction limit. The reduction limit is reached if a behaviour finishes executing or is blocked waiting on a communication from another behaviour. Decomposed behaviours will continue to communicate with behaviours outside the scope of the composition, insulating them from the changes.

3.7.1 Tools and Languages in the ArchWare Environment

The set of tools to support the ArchWare environment are (Oquendo et al., 2004):

- A UML based visual editor.
- A graphical animator which simulates the behaviour of an architecture based on cases defined to validate the architecture.
- A Model Checker.
- A Theorem Prover.
- A Refiner to assist in generating a concrete architecture from an architectural style.
- A code generator, which takes the code generation rules of a target language and produces a compiler from the ArchWare ADL to the target language.

A set of languages is also defined:

- The ArchWare ADL defines executable software architectures as a set of behaviours and connections (Sec. 3.7.4).

- The Architectural Analysis Language is used for the definition and verification of constraints on architectural styles. It is a formal property expression language designed to support automated verification. It allows specification of both structural and behavioural properties of a software architecture by combining predicate and temporal logic. The Model Checking tool is used to determine whether an architecture defined in ArchWare ADL satisfies the properties of a style defined in the Architectural Analysis Language.
- The Architectural Refinement Language guides development of a correct architectural description by ensuring that, in an incrementally refined architecture, refinements are correct according to a set of refinement conditions.

3.7.2 ArchWare Runtime

The ArchWare Runtime supports deployment and runtime evolution of executing software architecture descriptions. Its components are an ArchWare ADL VM, a compiler from the ArchWare ADL to the VM, and ArchWare ADL Hypercode. Together these components provide the mechanisms to support evolution. Much of the work described in this chapter has contributed to the ArchWare Runtime.

The compiler translates ArchWare ADL into ProcessBase, which is then compiled by a ProcessBase compiler, and executed on the ProcessBase VM. Therefore, the ArchWare ADL VM is implemented using ProcessBase.

The ArchWare ADL Hypercode system runs as part of the VM. The compiler accepts Hypercode. Elements of the executing system can be inspected using the Hypercode Editor, which displays a Hypercode Representation of any first class value in the language. Structural reflection is used in combination with the Hypercode system to evolve the executing software architecture.

The VM supports compliance by providing mechanisms for feedback from the executing system (Balasubramaniam et al., 2004b). Probes can be inserted in the VM and gauges are defined to interpret the information collected by the probes. The probes are flexible and therefore suitable for inclusion in an evolving system. Both are implemented in the ArchWare ADL and were described in Section 1.1.1.

3.7.3 Process Models

Process Models, written in the ArchWare ADL, co-ordinate software architecture design, analysis and evolution in the ArchWare environment. They are used to define a software architecture based on a particular architectural style and then refine it from an abstract to a concrete representation, whilst preserving the style constraints. In addition, Process Models co-ordinate the tasks of developers and the various tools they use in the development and evolution of an architecture.

Process for Process Evolution (P2E) technology uses meta processes, which exploit the architectural description of the system in controlling its own evolution. A system is built hierarchically from components consisting of behaviour pairs, as shown in Fig. 3.6. One element in the pair is the *Producer*, which is responsible for the main functionality of the component. In the diagram it takes in raw values and produces widgets. The other element is the *Evolver*, which manages the Producer's evolution. The Evolver is connected to the Producer by probe, feedback and change connections (Balasubramaniam et al., 2005; Balasubramaniam et al., 2004b). Fig. 3.7 shows how P2E Components are composed into a system. Producer and Evolver are explained further in Appendix E.

3.7.4 ArchWare ADL

ArchWare ADL is a computationally complete architecture description language developed as part of the ArchWare project. Providing direct implementation support for software architecture represents a departure from the normal approach, which decouples the software architecture from the implementation. The advantages of integration are delineated in papers on ArchJava (Aldrich et al., 2002c; Aldrich et al., 2002a), another system which integrates the software architecture and the implementation:

- Software architecture information is directly accessible in the source code.
- Conversely, the implementation is accessible from the software architecture so a developer can see how a particular component is implemented.
- Views of the software architecture are automatically extracted from the implementation and are always up to date.

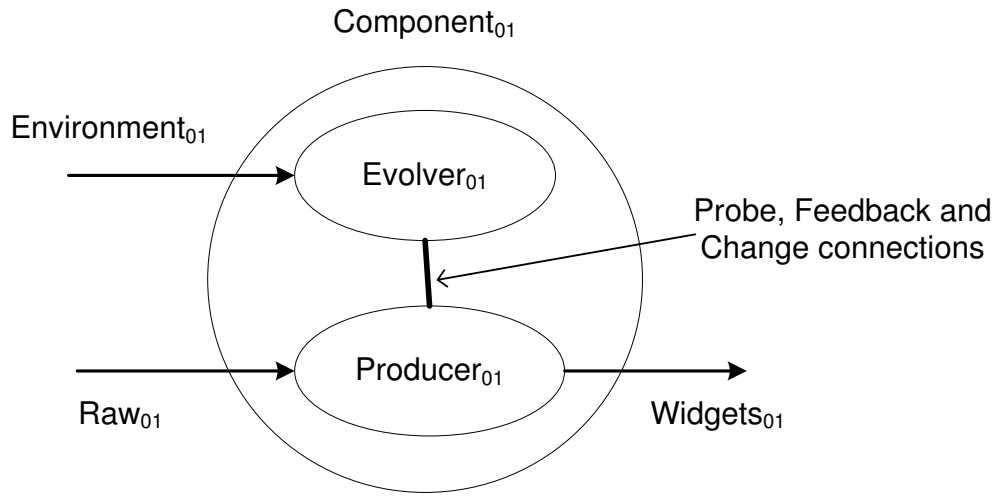


Figure 3.6. P2E Element. A P2E Component is composed from an Evolver and a Producer.

- Inconsistencies between the software architecture and the implementation are avoided.
- The implementation is guaranteed to obey architectural constraints. It follows that analysis of the software architecture is relevant to the implementation.
- The architecture and code remain consistent as they evolve.
- The type system checks communication between components, as opposed to developers following guidelines that are not automatically enforced.

There are also some drawbacks to a computationally complete architectural description language. Firstly, developers are limited in their choice of language which can impede the take up of the technology. Secondly, a stand alone architecture description may have a number of alternative implementations for different platforms and environments. The possibilities for generating multiple implementations are constrained by merging the architecture and the implementation.

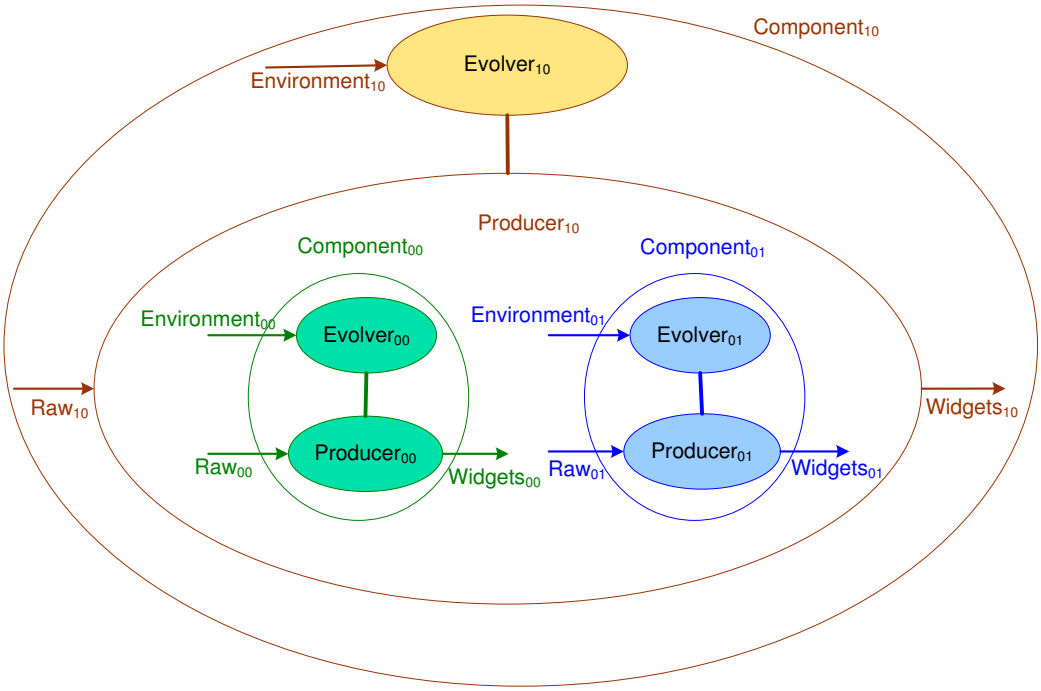


Figure 3.7. Hierarchical P2E Components. P2E Components are structured into hierarchical systems.

The ArchWare ADL has a formal foundation based on the higher order π-Calculus (Milner, 1999). It is a strongly typed architectural description language with styles layered on top, which provides executable specifications of evolvable systems. Its novelty is in the combination of π-Calculus, persistence, decomposition, reflection, reification and Hypercode. The three principles of abstraction, correspondence and type completeness (page 83) were used in the language design. The language guarantees syntactically consistent changes, because all changes are compiled and type checked before being reflected into the system.

The change mechanisms available in the ArchWare ADL to enable incremental evolution are (Balasubramaniam et al., 2004b):

- A decomposition operator to incrementally break up the system (Sec. 5.3.1).
- A representation of executing code used for introspection (Hypercode).

- Structural reflection provided by a callable compiler with a *coerceToValue* function.
- Type system introspection provided by the *typeOf* function.
- An infinite union type (*any*) with injection and projection operations.

3.7.5 Dynamic Change in ArchWare

The kinds of change in an ArchWare environment are:

- Replacement of components.
- Static and dynamic generation of new components.
- Dynamic evolution, involving decomposition, reification, reflection and recomposition.

An ArchWare system is built by composing sets of communicating behaviours. Composition unifies communication channels between the behaviours and returns a handle to the composed set. A behaviour is a unit of functionality and can communicate with other behaviours through its connections. Any first class value in the language can be sent along a connection. This includes behaviours as well as connections. Dynamic interactions between components are created by unifying connections, using a *unify* operator, inside a composition clause.

Decomposition is the language mechanism for partially stopping an executing system. It is the inverse of renaming in the π-Calculus. On decomposition, the set of composed behaviours will be disconnected, but will continue to execute up to their respective reduction limits. That is, connections which were *unified*, or connected, in the composition will be broken, and the components will continue to execute until they either finish or are blocked waiting on a connection. In practice, this involves the suspension of the targeted threads in a multi-threaded system, while the rest of the system continues executing. Behaviours should be programmed so that the reduction limit semantics cause them to do something sensible when they are decomposed, for example, waiting until their communication channels are reconnected, or stopping. The result of a decompose operation is the set of behaviours in the composition.

Compose and Decompose

Composition in the ArchWare ADL gives a single handle to a number of behaviours. Behaviours may spawn new behaviours but these are not directly accessible from the composition. Inside the compose statement, the connections between the behaviours may be unified, allowing them to communicate with each other.

A compose statement in the ArchWare ADL returns a behaviour. An example of the use of compose, taken from the ArchWare ADL reference manual (Balasubramaniam et al., 2004a) is given in Fig. 3.8. A behaviour, `b1`, is defined which sends the value `100` along channel `c1` and then receives a string on channel `c2`. Another behaviour, `b2`, is defined which receives an integer on channel `c3` and then sends a string on channel `c4`. The `compose` statement gives a single handle to the two behaviours and unifies their channels. In the first line of the `compose`, `b1` is included in the composition and given the label `send_int_receive_string` for the scope of the `compose` statement. In the second line, `b2` is included and renamed similarly. The `where` clause of the composition connects the communication channels between the behaviours. `c1` is connected to `c3` and `c2` is connected to `c4`. Once the compose statement has been executed, the behaviours will communicate with each other along the connected channels. The `composed_system` is itself typed as a behaviour.

Decomposing breaks the unification of the connections which were unified in the composition. The decomposed behaviours will continue to execute until they reach their reduction limits. This means that the processes will continue to execute until they are waiting for a communication from another process, or they may simply finish executing. A decompose statement acts on a behaviour and returns a sequence of the behaviours which were used to create the composition. The behaviours in the sequence have the same ordering as in the compose statement.

An example of decompose, from the reference manual, is shown in Fig. 3.9. In the first line, the `composed_system` is decomposed and the result assigned to `behaviour_seq`. The second line shows how the behaviour `b1` can be accessed from the sequence. The final line shows how its label can be accessed. The label was associated with the behaviour in the original compose statement and can be used to identify it after decomposition.

Table 3.1. ArchWare. This table categorizes the support for evolution in ArchWare.

Dynamic Change

Openness	ArchWare supports the creation of new component types and instances to introduce new functionality.
Maintain State	Maintaining hyperlinks in a Hypercode graph representation allows program state and internal data to be preserved over change.
Connections Bound and Unbound	In the ArchWare ADL connections can be bound in a composition and later unbound using decomposition.
Reflection	Structural reflection is available to introduce new values into the executing system.

Autonomous Change

Automation	No
Automated Motivations for Changes	Constraints in the Architectural Analysis Language as well as flexible probes and gauges.
Automated Support for Openness	The MPF allows new functionality to be automatically introduced into the system.
Model Available to Management Components	Hypercode graphs provide a model of any value in the system and allows introspection of current program state.

Change Localization

Higher Order Connectors	No
Programmable Connectors	No
Partial Decomposition	Yes - the decomposition operator can stop a small part of the system so it can be evolved.

Change Management

Change Management	Process models contain the change management policy.
View of Executing System	No
Partial Views	Yes - Hypercode graphs.
Vocabulary for Specifying Changes	Changes may either be specified as a set of MPF operations on a Hypercode graph or else defined in the ArchWare ADL.
Open Adaptation	Yes
Closed Adaptation	Yes
Architectural Styles	The Architectural Analysis Language is used to define styles.

```
value b1 = behaviour {
                    via c1   send 100
                    via c2   receive s : string
                  }
 value b2 =  behaviour {
                    via c3   receive i : integer
                    via c4   send 'Hello'
                  }

 value composed_system = compose {
     send_int_receive_string   as b1
      and receive_int_send_string   as b2
       where {
        send_int_receive_string ::c1   unifies receive_int_send_string ::c3 ,
        send_int_receive_string ::c2   unifies receive_int_send_string ::c4
       }
}
```

Figure 3.8. ArchWare ADL compose statement. Composing two behaviours and unifying their connections.

```
value behaviour_seq  = decompose composed_system
value b_value1  = behaviour_seq::1.bhvr          ! behaviour denoted by b1
value b_label2 = behaviour_seq::2.label          ! label of b1
```

Figure 3.9. ArchWare ADL decompose statement. Decomposing the composed behaviours.

3.8 Hypercode for the MPF

Orthogonal persistence, structural reflection, ProcessBase and the ArchWare ADL have been used to build a Hypercode system for use as an experimental base in the MPF. A new implementation technique makes Hypercode reusable and augments it with an interface that allows Hypercode operations to be accessed by other programs.

Hypercode for the MPF can be transferred with minimal difficulty to provide the same functionality for different programming languages, as long as they support structural reflection, referential integrity and first class code. The transferral is made possible by using algorithms for the Hypercode operations that are independent of language syntax. These algorithms rely on Hypercode constructs that are inserted into programs using mark-up. It is not possible to have a completely language independent Hypercode system because firstly, evaluation requires compilation and secondly, the explode operation involves generating representations of program values in the programming language. Therefore, transferring Hypercode to a new language necessitates plugging in a new compiler and changing the generative code in the explode operation to produce the new language.

Initially, the Hypercode system was implemented in and for ProcessBase because it provides orthogonal persistence, ensuring hyperlinks remain valid for their lifetime, and structural reflection, which is needed for implementation of the evaluate and explode operations. Then, in order to provide support for an incremental evolution process in the MPF, the Hypercode system was transferred to ArchWare ADL. This allows Hypercode programs to be written that take advantage of the decompose operator to partially stop an executing system. The ArchWare ADL Hypercode system has been used as part of the Tower Browser described in Appendix E.

An interface to the Hypercode system exposes the Hypercode operations and facilitates the development of other programs that use Hypercode. The interface is used by the MPF to enable meta-programs to explode and evaluate Hypercode graphs. In addition, a user-interface relies on the Hypercode operations to permit developers to program and evolve Hypercode.

3.9 Summary

This chapter gives details of a set of technologies on which the MPF is based.

Orthogonal Persistence A persistent store provides referential integrity for the MPF's program representations that include both source code and data.

Structural Reflection The creation and introduction of new values during execution is made possible using structural reflection.

Hyper-Programming and Hypercode Hyper-programming used hyperlinks to include extant values in program source code. Hypercode extended the concept to abstract over the difference between code and data and provide introspection on an executing system.

Entity and Representation Domains Program representations, e.g., Hypercode, and their corresponding entities, e.g., executing programs and data, can be considered to reside in separate domains. The concept of entity and representation domains supports reasoning about the mappings between entities and their representations.

Hypercode Operations The *explode*, *implode*, *evaluate* and *edit* operations define the interface to Hypercode.

ArchWare The MPF was developed as part of the ArchWare project, which provides the ArchWare ADL, a technology to model software architectures. The composition and decomposition operators in the ArchWare ADL facilitate the partial decomposition of executing programs, thereby supporting incremental evolution. The ArchWare project has produced other tools and mechanisms for evolution, including an environment for process evolutions that performs change management.

MPF Hypercode A new Hypercode system has been developed for the MPF that is easily re-used and has an interface that allows other programs to use the Hypercode operations. It also implements Hypercode for ArchWare ADL.

Chapter 4

Hypercode Graphs and the MPF Operations

The MPF provides two mechanisms for evolution: a model that meta-programs can use to represent an executing system; and a set of transformations on the model to describe evolutionary changes. The model is the Hypercode graph and the transformations are implemented by the MPF operations.

The MPF provides an environment in which management components in the form of meta-programs can access representations of values in an executing system. The representations, called Hypercode graphs, can be examined to determine the current state of internal program values. Meta-programs can also use them for evolution in a process that involves changing a Hypercode graph and then reflecting the altered graph into the executing system.

An executing program can be represented by Hypercode graphs because they characterize both program closure and internal state. Program code is merged with program data using hyperlinks. They also include program syntax to give meta-programs evolving a graph the full power of the programming language. The MPF defines Hypercode graphs and an API, or set of operations, for meta-programs to interact with them. The operations allow meta-programs to *traverse*, *manipulate* and *evolve* the graphs.

A meta-program, starting with a handle to a value in the form of a hyperlink, can dereference the hyperlink to obtain a Hypercode graph representing the value. It may then, for example, traverse the graph to discover some internal state of the value stored in a variable. The traversal involves using MPF operations to find the hyperlink that references the variable using its name or position in the program. By following the hyperlink, the meta-program can access the variable's value.

In order to make changes to executing systems, meta-programs use the MPF operations to manipulate Hypercode graphs. For example, a meta-program can update a hyperlink to reference a different value. Alternatively, it could alter a function to make it use a different algorithm. In the latter case, the Hypercode graph representation of program closure would allow the algorithm to be changed without the function losing its internal state. These examples illustrate some possible manipulations, but the actual changes a meta-program can make are only restricted by the programming language being used.

After the changes, the process used by a meta-program evolving the system involves evaluating the Hypercode graph. The result of evaluation is a new value, possibly containing references to existing values, which can be incorporated into the executing system. The way in which this is done depends on the change management structure in the system. The new value may, for example, replace the old one in an updatable location.

4.1 Hypercode Graphs

A Hypercode graph is a combination of an abstract syntax tree and a data graph. This structure provides a meta-program with a complete program representation encompassing both closure and syntax. Any part of an executing program can be accessed through traversal of the graph representation from its root.

4.1.1 Structure

Abstract syntax trees (Aho et al., 1986) are a standard format for internal program representation (Czarnecki and Eisenecker, 2000). However, an abstract syntax tree by itself is not sufficient to represent Hypercode, which encompasses not just program syntax but also program closure. Therefore, the program structure provided by the MPF combines an abstract syntax tree with a data graph. A similar approach is taken in Intentional Programming, where a program is represented by a set of intentions and an intention instance may refer to program data or program syntax (Sec. 2.7).

A Hypercode graph appears to a meta-program as an abstract syntax tree with references to data values at appropriate points in the tree. For example, in a clause where a value is assigned to an identifier, the value may be a reference to the data graph. Following one of these references gives a meta-program access to a Hypercode graph representation of the value. The meta-program can also access the value itself.

A meta-program traversing a Hypercode graph encounters two types of links, illustrated in Fig. 4.1. Firstly, links in the abstract syntax tree, which is a subset of a Hypercode graph, shown in black in the diagram, reference structures in the program source. For example, an `if` statement references its condition and its branches. These are standard types of links which would be available to any meta-program dealing with an abstract syntax tree representation.

Hypercode provides the second type of links, which are hyperlinks, coloured blue in the diagram. Hyperlinks are pointers from syntax tree nodes to the data graph. They allow a meta-program access to the program closure. To traverse the data graph, a meta-program dereferences a hyperlink to access either another Hypercode graph, or the value it represents. If Hyperlinks were not included in Hypercode graphs, they would have a tree structure. The overall structure is a graph because hyperlinks give access to the data graph and values in the data graph may include references back to the program itself.

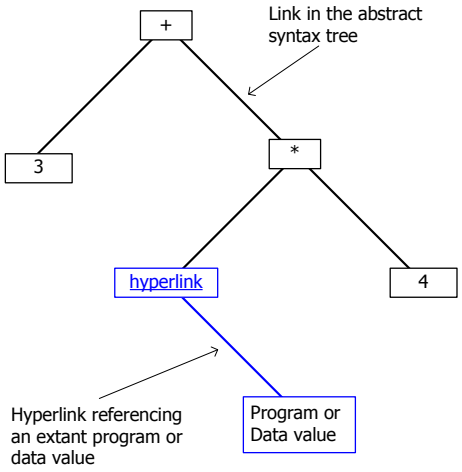


Figure 4.1. Link Types. The two types of links in a Hypercode graph representing the code fragment: 3 + hyperlink * 4.

4.1.2 Type

Figure 4.2 shows the basic structure of a Hypercode graph in the MPF. It is a bi-directional graph where nodes have a parent - child relationship. Each node has one parent and zero or more children, except for the root node, which has no parent. Nodes may contain either *program syntax* information, *program text* or a *hyperlink*. Consider the example of a Hypercode graph representing $4 + 3$ in Figure 4.3. The top node (clause) contains syntax information, as do the nodes in the middle row. The nodes at the bottom contain program text, which is the same as program source code. This type of node is always a terminal node with no children.

The previous example did not include a hyperlink, which is the third type of node. Figure 4.4 shows a graph representing an expression, where 3 is added to an extant value referenced by a hyperlink. The hyperlink node, shown in blue, has the name *hyperlink* as well as an ID attribute. When a meta-program dereferences the hyperlink, the ID is used by the MPF to find the associated value. The level of indirection allows a Hypercode graph representation to retain its validity outside the MPF. The graphs can therefore be used by other tools, such as the user interface (Sec. 7.5). Note that the graph does not show the value of the integer referenced by the hyperlink, this is because the meta-program needs to explicitly dereference the hyperlink to access it.

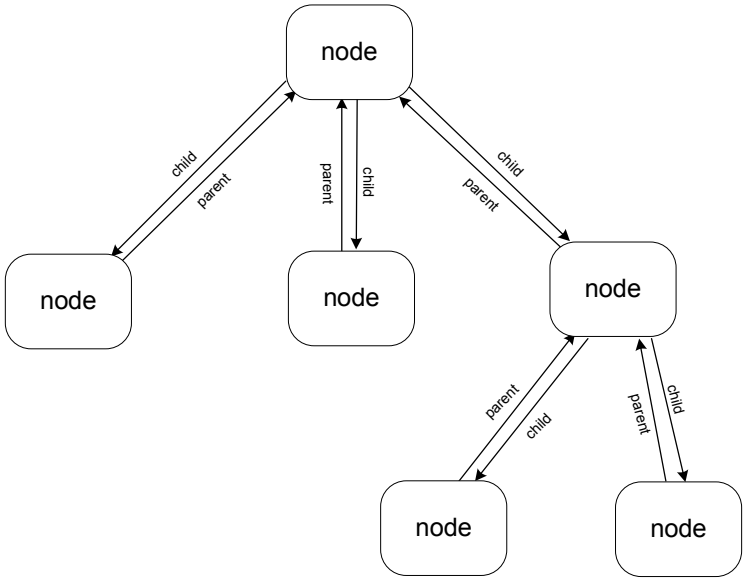


Figure 4.2. Hypercode Graph. The Hypercode graph is a bi-directional graph, where nodes have a parent - child relationship.

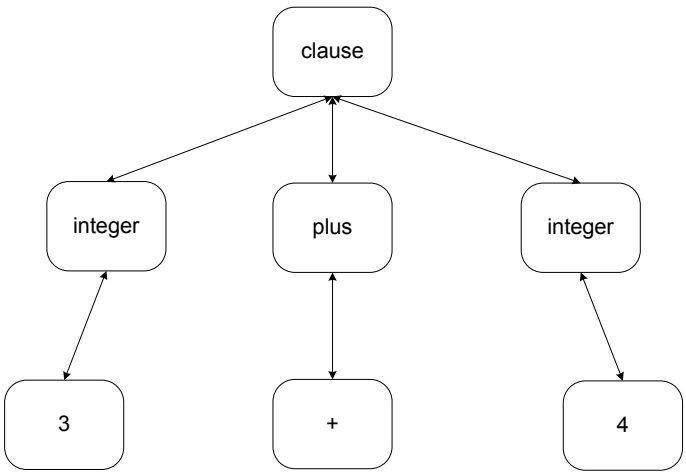


Figure 4.3. Hypercode Graph Example. This Hypercode graph represents the expression: $4 + 3$, showing nodes containing program syntax and nodes containing program text

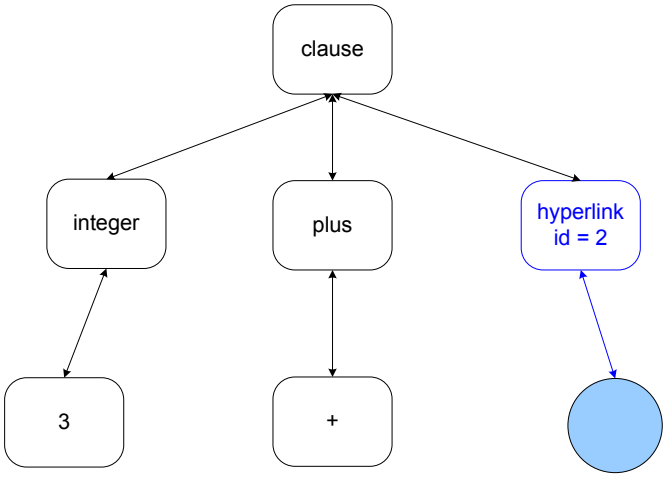


Figure 4.4. Hypercode Graph with Hyperlink. A Hypercode graph representing an expression where the integer 3 is added to a value referenced by a hyperlink.

A pseudocode definition of the graph type in Figure 4.5 shows that a graph is a record containing a node, a parent and a list of children. The ordering of the list of children is important because it represents the order of statements in the program.

```
type graph is record
        (
          node: nodeType,
          parent: graph,
          children: list of type graph
        )

type nodeType is record
        (
          name: string,
          text: string,
          attributes: list of (name, value) pairs
        )
```

Figure 4.5. Hypercode Graph Definition. The type definition of a Hypercode graph.

Nodes, defined by nodeType in Figure 4.5, are records with three parts: a name, text and a list of attributes. The form of a node instance depends on the type of information it contains.

Program syntax nodes store a string in the *name* field, e.g., `'integer'` (Fig. 4.6).

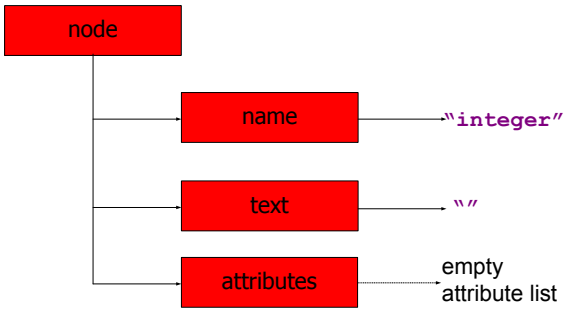


Figure 4.6. Program Syntax Node. A program syntax node for *integer*.

Program text nodes store a string in the *text* field, e.g., `'3'` (Fig. 4.7).

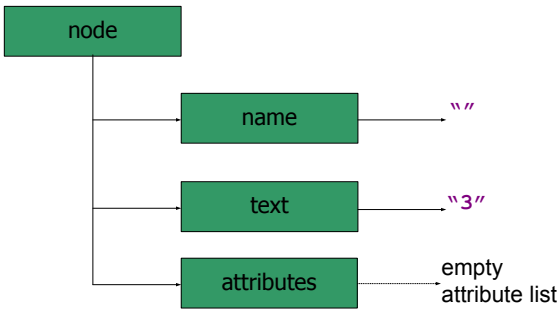


Figure 4.7. Program Text Node. A program text node for *3*.

Hyperlink nodes store the string `'hyperlink'` in the *name* field and their ID in the *attributes* field (Fig. 4.8).

4.1.3 Example

A Hypercode graph representing a record of personal and work information is illustrated in Figure 4.9. The first part of the record contains a personal name, address and email contact. The second part contains a company name, address and email. The

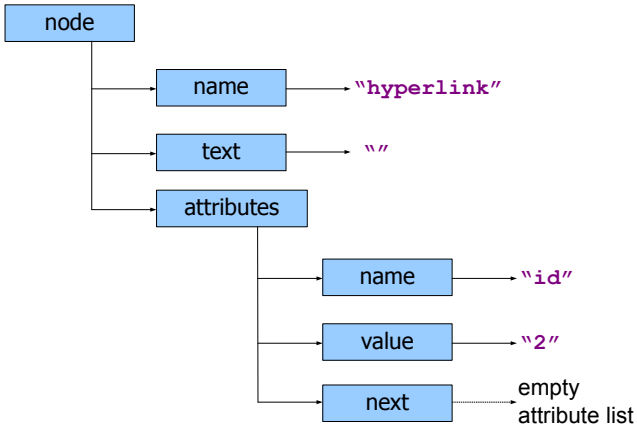


Figure 4.8. Hyperlink Node. A hyperlink node with an ID value of 2.

data values, such as the person's name, are all referenced by hyperlinks. This example portrays how the abstract syntax tree part of the Hypercode graph (simplified in the diagram) is connected to the data graph (circles) by hyperlinks. It also shows how multiple hyperlinks can reference the same data value. In this case the person's personal and work email addresses are the same. Because they are able to model sharing in this way, Hypercode graphs can represent program closure.

4.2 Operations

The MPF operations provide an interface to Hypercode graphs through which meta-programs can traverse, manipulate and evolve them. The operations for traversal allow meta-programs to locate parts of a Hypercode graph relevant to their purpose and discover the current state of internal program values. Operations for manipulation facilitate changes to existing Hypercode graphs and the creation of new ones. After changes have been made, meta-programs use the operations for evolution to incorporate the changes into the executing system.

The definitions of all MPF operations are presented in two libraries, defined in Appendix B.

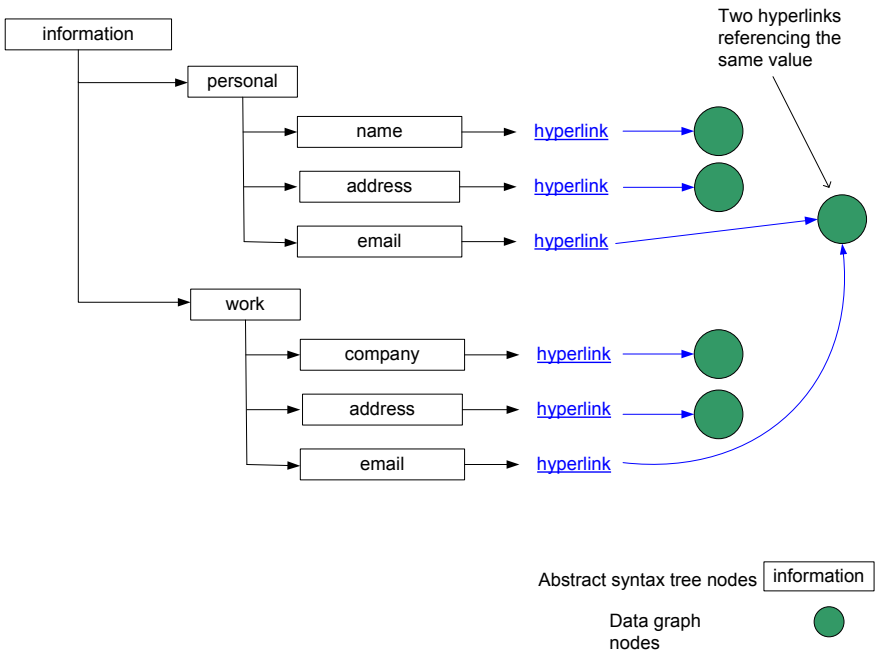


Figure 4.9. Hypercode Graph. A Hypercode Graph incorporates program syntax and existing values.

4.2.1 Traverse

Meta-programs traverse Hypercode graphs by following program syntax links to reach syntax nodes and following hyperlinks to access function and data values. The full list of operations for traversing Hypercode graphs is defined in Tables 4.1 and 4.2.

Operations that act on program syntax nodes (Fig. 4.1) are standard operations, which could be performed on any tree, for example *getChildNodes* and *getParentNode*. The latter is applied to nodes in the program syntax tree, which only ever have one parent. It cannot be applied to values, which do not store information about references that point to them.

Operations that act on hyperlink nodes allow meta-programs to access values from the data graph. For example, *getGraphFromHyperlink* dereferences a hyperlink to return a Hypercode graph representation of the value. Figure 4.10 depicts an example of it being applied to part of the Hypercode graph from the example in Figure 4.9. In step 1, the meta-program has a handle (shown as an arrow from the meta-program cloud) to the hyperlink referencing the address part of the personal information. The *getGraphFromHyperlink* operation is applied to the hyperlink in step 2 and returns a

Table 4.1. Traversal Operations. The operations available for traversing Hypercode graphs.

Name	Description
getAllHyperlinks (graph) → graph list	Get all the hyperlinks in the graph (in-order depth first search).
getAllNodesWithName (string, graph) → graph list	Get all the nodes with the given node name in the graph (in-order depth first search).
getAttributes (graph) → attributes	Get the list of attributes from a graph node.
getAttributeValue (string,graph) → string	Get the value of an attribute, given its name.
getChildNodes (graph) → graph list	Get a node's list of children.
getChildPosition (graph) → integer	Get the position of this graph node in the list of children of its parent, as an integer.
getFirstChild (graph) → graph	Get the first child of this node.
getFirstChildWithName (string,graph) → graph	Get the first child of this node with the given node name. Only searches the children of this graph node.
getFirstHyperlink (graph) → graph	Get the first hyperlink that is a descendent of a graph node, searching in program source code order.
getFirstTagWithName (string,graph) → graph	Get the first program syntax node with the given name that is a descendent of the given graph, searching in program source code order. Searches the whole graph descending from this node.
getGraphFromHyperlink (graph) → graph	Get the graph representation of a value referenced by a hyperlink.
getGraphFromValue (any) → graph	Get the graph representation of the given value.
getLastChild (graph) → graph	Get the last child of this node.
getLocalName (graph) → string	Returns the name of this node. Only valid for program syntax type nodes.
getNextSibling (graph) → graph	Get the node immediately following this node in the list of its parent's children.
getNodeName (graph) → string	If this node is a program syntax node return its name. If it is a program text node return the text. For a hyperlink node return `hyperlink`.
getNodeType (graph) → string	If this node is a program syntax node or a hyperlink return the string: `element`. Otherwise return the string: `text`.
getNodeValue (graph) → string	If this node is a program text node get its text otherwise return an empty string.
getNumberOfChildren (graph) → integer	Get the number of children of this node.
getParentNode (graph) → graph	Get the parent node of this node.
getPreviousSibling (graph) → graph	Get the node immediately preceding this node in the list of its parent's children.

Table 4.2. Traversal Operations cont. The operations available for traversing Hypercode graphs.

getValueFromHyperlink (graph) → any	Get the value referenced by a hyperlink (as an infinite union type).
hasAttributes (graph) → boolean	True if this node is a program syntax node or a hyperlink and it has some attributes.
hasChildNodes (graph) → boolean	True if this node has any children.
isEqualNode (graph,graph) → boolean	Compares two graphs to see if they represent the same program.
isSameNode (graph,graph) → boolean	True if two graph nodes are the same node.
getXMLFromGraph (graph) → string	Return graph represented as a string of XML.

Hypercode graph representing the address. This new Hypercode graph is shown in step 3, where the meta-program has a handle to its root. To find out the current address, the meta-program then uses *getGraphFromHyperlink* on the first hyperlink in the new graph. This returns the string containing the street part of the address shown in step 4. By following hyperlinks, as in this example, meta-programs can access internal program state and values.

Instead of dereferencing a hyperlink to obtain a Hypercode graph, a meta-program may obtain the actual value referenced by the hyperlink. That is, traversal operations over hyperlinks allow a meta-program to retrieve a value in two forms. Either the value itself may be accessed, e.g., an executing program, or its representation, e.g., the value's associated Hypercode graph. The *getValueFromHyperlink* operation takes a hyperlink node and returns the value it references. The Hypercode graph of any value can be obtained using the *getGraphFromValue* operation without needing a hyperlink that references the value.

The four operations *getChildNodes*, *getParentNode*, *getGraphFromHyperlink* and *getValueFromHyperlink* are all that is necessary for a meta-program to be able to traverse both the syntax and the data parts of a Hypercode graph. The other operations that have been defined are syntactic sugar and provide convenience.

A program's closure is characterized by the data graph that is accessible through hyperlinks in a Hypercode graph. The *getAllHyperlinks* operation can be used by a meta-program to gather the set of values representing the closure.

The *getXMLFromGraph* operation produces XML representations of Hypercode graphs, allowing them to be exported and, for example, depicted in a user interface.

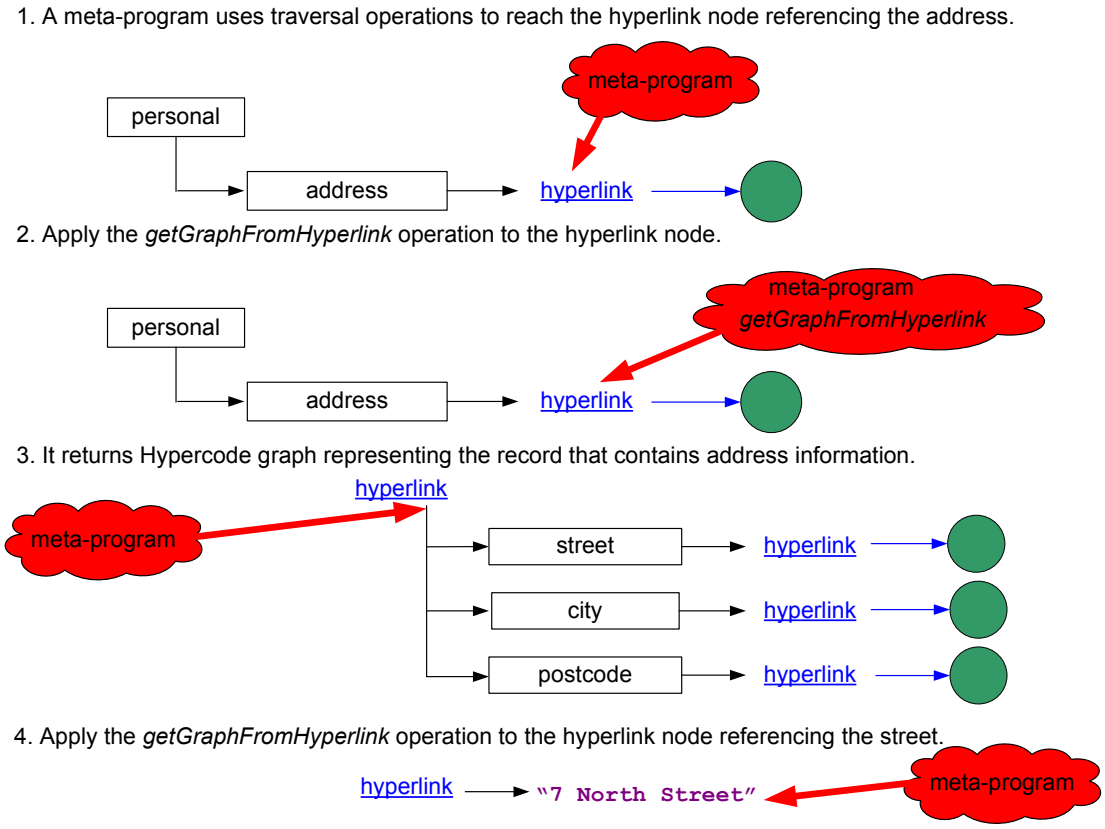


Figure 4.10. The getGraphFromHyperlink Operation. *getGraphFromHyperlink* dereferences a hyperlink to return a Hypercode graph representation of the value.

4.2.2 Manipulate

Operations to manipulate Hypercode graphs create, delete, add and remove nodes. Meta-programs have the full power of the programming language when creating and changing graphs. In addition, they can link extant values into the program by creating new hyperlink nodes and update existing hyperlinks to reference different values. The full list of operations for manipulating Hypercode graphs is defined in Table 4.3.

By manipulating Hypercode graphs, meta-programs can, for example, change the representation of an existing function. Figure 4.11 illustrates a function (1) that calls *foo*, passing it a hyperlinked counter value. The function is changed (2) to include a clause incrementing the counter. Part 3 of the diagram portrays a Hypercode graph of the function before the changes. The new clause (4) is added at the start of the function using *appendChildAt* position 0, appending it to the start of the function's main clause.

Table 4.3. Manipulation Operations. The operations available for manipulating Hypercode graphs.

Name	Description
addAttribute (string,string,graph)	Takes an attribute pair (name, value) of strings and adds it to a graph's list of attributes. For program syntax and hyperlink nodes only.
appendChild (graph,graph) → graph	Adds a child to the end of the list of children of this graph and returns the graph. If the graph is nil, the child is returned.
appendChildAt (graph,graph,integer) → graph	Adds a child at the given position in a graph's list of children and returns the graph. If the graph is nil, the child is returned.
appendChildDescend (graph,graph) → graph	Adds the child node to the end of the list of children of the graph and returns the child.
copy (graph) → graph	Create a copy of a graph.
copyHyperlink (graph) → graph	Create a copy of a hyperlink that can be inserted into a different graph.
insertBefore (graph,graph) → graph	Inserts a graph before the given graph in the list of its parent's children. Returns an empty graph if the child is not found in the list.
makeGraph (string) → graph	Construct a Hypercode graph representation of a program from an XML representation. Inverse of *getXMLFromGraph*.
newDeclarationGraphForGraph (string,graph) → graph	Create a graph of a value declaration, where the value is represented as a graph. (Takes a name and a graph.)
newElement (string,attributes) → graph	Creates a new graph node with a name and list of attributes.
newHyperlinkElement (any,string,graph) → graph	Create a new hyperlink node.
newTextElement (string) → graph	Creates a new program text graph node.
pasteHyperlink (graph,graph)	Paste a copied hyperlink (first parameter) into a new graph as a child of the given node (second parameter).
removeChild (graph)	Removes this graph from the list of its parent's children.
replaceChild (graph,graph)	Replaces the first instance of a graph in the list of its parent's children with the given graph.
setAttributeValue (string,string,graph)	Set the value of an attribute or create the attribute if it does not exist. Only applicable to program syntax and hyperlink graph nodes.
setNodeValue (string,graph)	Set the text value of this graph node, only applicable to program text nodes.
updateHyperlinkLocation (graph,graph)	Update a mutable location referenced by a hyperlink.

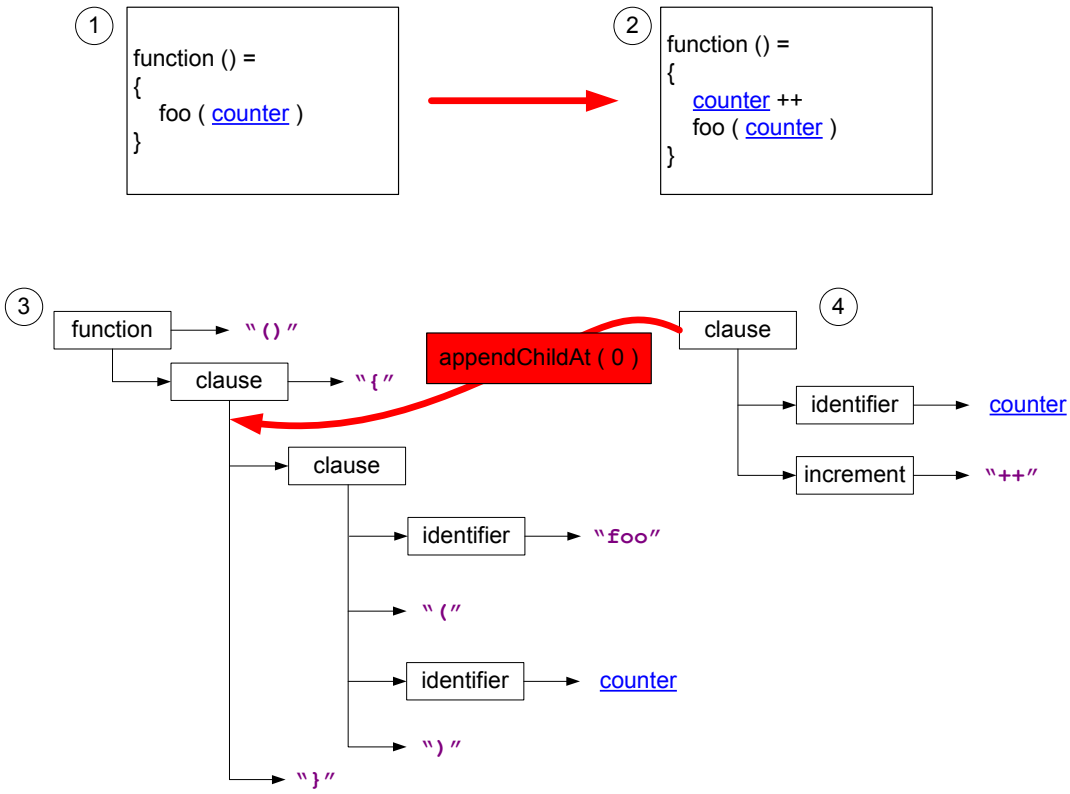


Figure 4.11. appendChildAt. Add a new clause into a function using the *appendChildAt* operation.

A meta-program can update the value in a location referenced by a hyperlink. An example is shown in Figure 4.12, where the *updateHyperlinkLocation* operation is used to change a person's occupation from a green value to a red value. The difference in colour represents that the value has changed. Using the *getGraphFromHyperlink* operation, the hyperlink could be dereferenced to find out what the actual occupation is. In order for a meta-program to use this operation, the hyperlink must reference a mutable location.

4.2.3 Evolve

The single operation that supports evolution is *evaluate*, which takes a Hypercode graph and compiles and executes it before returning the result of the execution as a graph. Meta-programs use evaluation to create new values that can be introduced into the executing system in a variety of ways, such as using the `updateHyperlinkLocation`

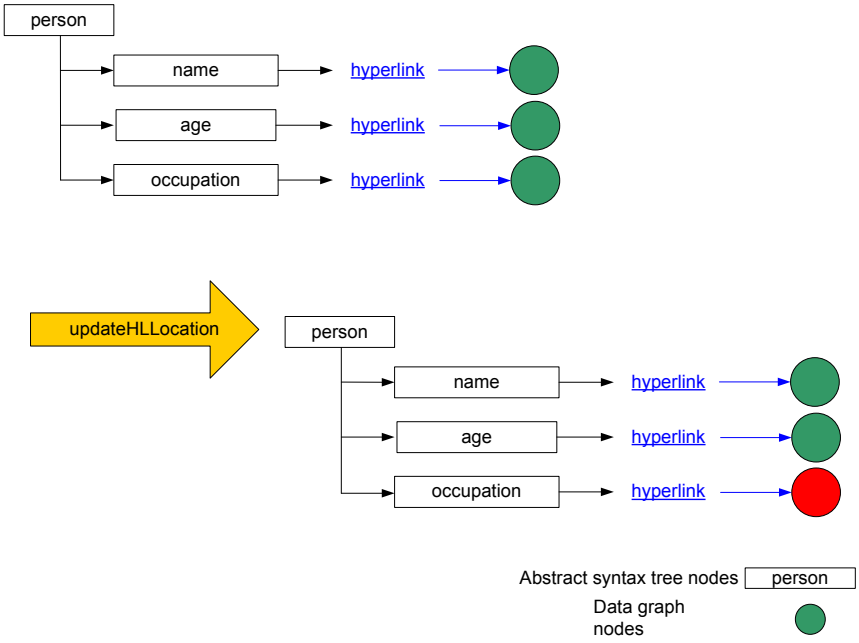


Figure 4.12. updateHyperlinkLocation. Update a value in a location referenced by a hyperlink.

operation to place a new value into a location accessed by an existing program. New values may also be incorporated with existing values using the incremental evolution process described in Chapter 5. In an evolutionary system, the technique used is ultimately decided by a change management policy implemented on top of the MPF.

4.2.4 Unified Representation

The MPF operations have been developed within the framework and are represented in the same way as the entities over which they operate. Therefore, a meta-program can evolve the MPF operations in the same way as any other value. A change management policy might involve creating new operations to add to the framework. For example, it is useful to create evolution patterns (Sec. 5.4) that support a particular evolution process and ease the difficult task of writing meta-programs.

4.3 Infosets and DOMs

An Information Set (Infoset) is an abstract description detailing the properties of XML documents (W3 Consortium, 2004). Definitions in an Infoset are used by XML technologies that examine, create or modify XML documents to describe what parts of the document they operate on. XML documents are defined as a set of *information items*. For example, *attribute information items* are the attributes in XML tags.

The Document Object Model (DOM) is a standard set of objects for representing XML documents, paired with a standard interface for accessing and manipulating them (W3 Consortium, 2005a). It is a logical model for the abstract interface defined by the XML Infoset, but not an implementation of that Infoset. The DOM may be implemented in any convenient manner.

The MPF can be compared with Infosets and DOMs, as summarized in Table 4.4. Where a DOM is a logical model for an XML document, a Hypercode graph is a logical model for a function or data value. The set of objects representing XML documents are equivalent to the parts of a Hypercode graph. The interface for accessing and manipulating the objects is equivalent to the MPF operations. In the same way that a DOM is a logical model implemented differently for different platforms, the MPF's operations and program representation can be applied to and implemented for different programming languages.

The XML Infoset is an abstract description for XML documents. Similarly, the MPF's program representation is an abstract description of executing programs consisting of a program model with syntax and data information and a representation of closure and sharing.

Table 4.4. Infoset, DOM and MPF Comparison. The parts of the MPF that are equivalent to Infosets and DOMs.

Infoset / DOM	MPF
Infoset	Program representation that can represent an executing program, represent program closure and data sharing, and be evolved.
DOM standard set of objects	Hypercode graph
DOM standard interface	MPF operations

4.4 Interface Design

The operations defined by the MPF support imperative meta-programming (Sec. 1.4) over Hypercode graphs. Evolution policy is defined by the scheduling of the operations decided by the meta-programmer and depends on the application environment. The MPF operations are sufficient for a meta-program to make any changes to existing values and create new values.

The design of the MPF is based on experience, from the ArchWare project, of evolving Hypercode through the user interface. In the user interface, the ArchWare ADL has been used in conjunction with Hypercode to produce a system that supports managed evolution through programmer interaction (Sec. E.1). Both the user-interface to Hypercode and the MPF use the same facilities to support the Hypercode operations *evaluate* and *explode* (Sec. 3.6.3).

Each operation implemented for the MPF has been chosen either because it has been shown to be useful for evolutions through the user interface, or else because it became apparent, after modelling examples in the MPF, that the operation was necessary to perform particular evolutions. As experience using the MPF is gained it is likely that the set of operations will be extended.

4.5 Framework Independence

The MPF has been developed as part of the ArchWare project and uses the ArchWare ADL. However, where possible, it has been implemented as a generic framework, which could be adapted to other languages. The program representation encompasses program closure by including both program syntax and data. This is a concept applicable to any programming language with the necessary supporting technology including structural reflection, referential integrity and first class functions. In addition, the operations for traversing and manipulating program representations could be applied for any programming language.

Some of the operations for evolution rely on the support of composition and decomposition mechanisms in the underlying language. ArchWare ADL provides these mechanisms for the implementation in this thesis, but they could be implemented in other languages. For example, decomposition can be modelled in Java using threads.

4.6 Summary

The MPF consists of Hypercode graphs that represent an executing system and a set of operations that meta-programs can use to traverse, manipulate and evolve the graphs. A Hypercode graph augments an abstract syntax tree with links to the data graph, characterizing program closure. Meta-programs have access to a Hypercode graph representation of any value, including themselves. Therefore, meta-programs can be evolved in the same way as other programs in the MPF. An example is given in Appendix D.

Chapter 5

Incremental Evolution

The Meta-Programming Framework supports the evolution of long-lived, complex systems. This chapter explains a process for these evolutions and how it fits in with the mechanisms available to support them.

In the process for incremental evolution that will be described, the entity performing an evolution (a developer or a meta-program) can reify parts of the executing program to provide a representation of the active system and its current state. This representation can be manipulated to apply unforeseen and unrestricted evolutionary changes and the altered representation can then be integrated into the running system.

5.1 Evolution Process

This thesis proposes the following principles, according to which a software system can be evolved in an ideal scenario:

The **Principle of Minimal Disruption** states that *evolution of a software system should proceed with minimal disruption to its execution*. From this principle it can be inferred that firstly, the evolution of part of an executing software system should entail minimal interference with the execution of the rest the program; and secondly, a component should be able to be evolved without losing its internal state and therefore be able to carry on after the evolution with the least possible disturbance.

The **Principle of Unconfined Evolution** states that *evolution should not be confined to a particular part of a software system*. Consequently every value in the system should be able to be evolved regardless of its type or size.

Applying the Principle of Minimal Disruption aims to minimize the ripple effect, where changes in one component cause additional changes in associated components. Complexity as defined by McCabe (McCabe and Watson, 1994) depends on components' fan-in and fan-out. Fan-in is a count of the calls to a component and fan-out is a count of calls from a given module. In systems with high complexity the ripple effect can be very disruptive and contribute to structural decay. This is a gradual drift away from the original design specification that can lead to the system becoming too expensive to maintain.

An evolutionary process that proceeds in accordance with these principles is illustrated in Figure 5.1 where an executing system in the bottom left corner is evolved by merging two of its components. The process has been published in the paper *Process Support for Evolving Active Architectures* (Greenwood et al., 2003).

Initially, the components in the figure have some internal state, represented by star shapes. The first step in the evolution is the *decomposition* of the executing system.

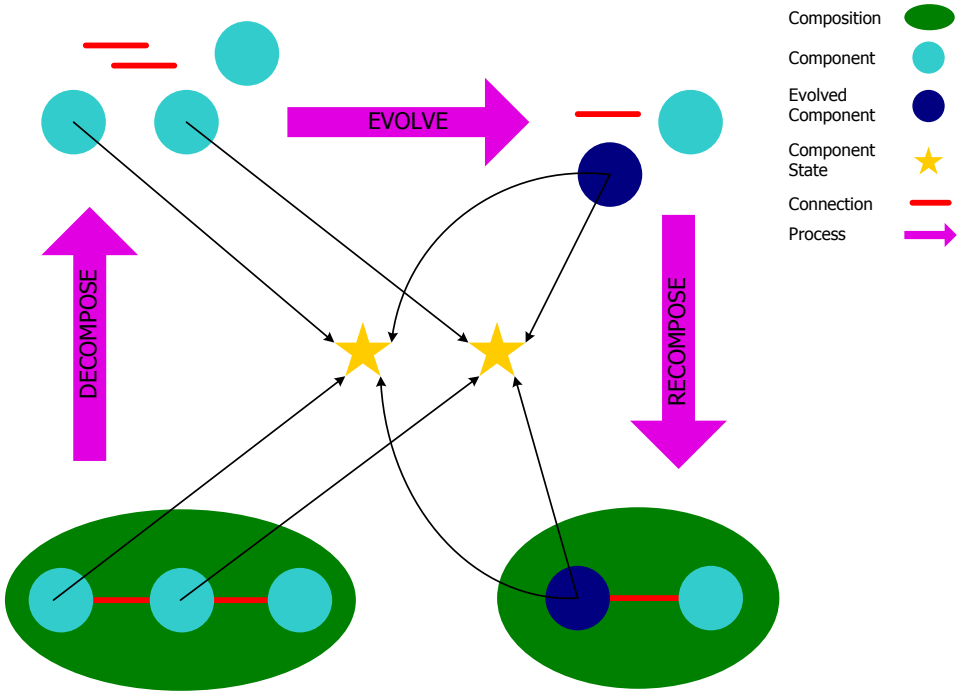


Figure 5.1. Evolution Process. A system is decomposed, evolved and then recomposed.

Decomposition stops the component that is decomposed but allows other components in the system to keep executing. The result of the decomposition is the set of sub-components that constituted the decomposed component.

After decomposition, a reification of the components provides a representation which can be manipulated to evolve them. In this example, the evolution merges two of the resultant components to form a single one that encompasses the internal state of its antecedents. The evolved representation must then be reflected to produce a value that can be integrated into the executing system. The components and connections are then *recomposed* to give an evolved system, which continues to execute using the same internal state as its predecessor.

Effecting incremental evolutions of this type requires:

- A means of halting the execution of those parts of the system involved in the change without stopping the whole system.
- A representation of the system to which changes can be applied.
- A mechanism to integrate changes into the executing system.

In the MPF the tools used to meet these requirements are decomposition, Hypercode graphs and structural reflection respectively. Decomposition has been implemented in the ArchWare ADL but could also be implemented in other languages, for example, in Java using groups of threads.

5.2 Tools for Evolution

In order to provide the mechanisms for the ideal evolutionary scenario we have:

- Decomposition to partially stop the system.
- A Hypercode graph representation of the active system that can be used by a developer or a meta-program.
- Structural reflection and first class functions to support reification, recompilation and rebinding.

Hypercode graphs can represent an executing program. The advantage of using them as a program representation is their ability to capture closure, allowing parts of a system to be represented after decomposition without losing state. Therefore, internal program data can be preserved over evolution by maintaining hyperlinks and evolution can proceed in accordance with the Principle of Minimal Disruption. The graphs provide a representation that can be used for both evolving the components and recomposing them into the new system. The compose and decompose operations used together with the Hypercode graph representation provide a powerful evolutionary tool.

Other properties of the framework for evolution are:

- A computationally complete language.
- Static and dynamic typing - programs which have been statically checked can be dynamically bound into the system.
- Components and connectors.
- Composition and decomposition with reduction limit semantics (Sec. 3.7.5).

An automatically evolving system requires a policy to determine how and when transformations are to be applied. The MPF provides the mechanisms for change, but does not determine evolution policy, which is application specific. The incremental evolution process described in this chapter is one possible process for which the MPF can be used. In the ArchWare Framework, process models provide a construction methodology and a process for controlled evolution (Sec. 3.7.3).

5.3 Incremental Evolution

The Principle of Minimal Disruption can be applied in the MPF context by a change management policy that defines an incremental evolution process. In this process, systems are decomposed at the level of individual components, giving a meta-program access to their Hypercode graphs. Using the MPF operation *getGraphFromHyperlink*, a meta-program can then explode the components to any level of granularity, conforming to the Principle of Unconfined Evolution. The Hypercode graph representation of the components can be altered by the meta-program to evolve the program whilst maintaining existing state and data. Finally, the changes are reflected into the executing system.

5.3.1 Composition and Decomposition

Hierarchical systems built using composition are suitable for evolution in the MPF context. Constructing the systems in this way allows decomposition to be used in the evolution of small parts of the system. The result of composing a set of components is another component that can be decomposed. A possible evolution, demonstrating the use of decomposition, is replacing one of the components in a composition, whilst retaining the others. After decomposition, a Hypercode graph gives a representation of the closure of the component being replaced. The graph can be used to construct an updated component with the same internal state. Using the MPF means the replacement can be automated.

In most programming systems, composition methods are irreversible and there is no controlled way of accessing the internal state and data of an executing program. Conventional measures to get around this involve either managing the information loss, or serializing the information and writing it out to the file system or a database. The

appropriate tools to do this must be put in place at development time. In some systems it may not be possible to break the encapsulation at all. In contrast to ad-hoc methods, the MPF supports a controlled and structured decomposition process.

5.3.2 Example of Evolution

To further explain the use of compose and decompose operations in an evolving system, an example is given following the scenario depicted in Figure 5.1. The original system, in the bottom left of the figure, is a composition of three filter components connected by two pipes. Two of the filter components refer to some data outside of the connected components, represented by the star shapes.

The first step for a meta-program evolving the system is to decompose it into its components. It starts with a handle to the system, which is a component in an updatable location. Decomposing the system gives the meta-program a Hypercode graph that references the filter components and stops communication between them. They may, however, continue to communicate with other components outside the composition. Note that the filter components still maintain their links to the data.

After the decomposition, two of the filters are combined into a single filter, creating a new component. The new component maintains links to the data referred to by the original filters. In order to effect the changes, the meta-program traverses the Hypercode graph resulting from decomposition to get the Hypercode graphs of the filter components. Using MPF operations, the graphs are combined to create the graph of a new filter with the same internal state as the filters it is replacing in the form of hyperlinks. Next, the meta-program uses the evaluate operation on the modified Hypercode graph to create a new filter value.

Finally, a new system is formed by re-composing the new filter component with the unchanged filter and the new system is placed in the updatable location. After the evolution, a new system has been created, but its composition includes, from its predecessor, the filter that was not replaced in the recomposition as well as existing state from the previous filters.

Figure 5.1 can be interpreted from both an architectural and a process perspective. From the architectural perspective, the diagram captures the structure of the current and evolved systems and the relationships between them in terms of which components are unchanged, modified or replaced. From a process perspective the diagram

captures how to evolve from the current to the new system: decompose into parts, replace some components and recombine in the new configuration.

This process would be part of the policy of a change management system. For example, an autonomic system, following the structure shown in Fig. 1.5, or a hierarchical system as in the ArchWare environment (Greenwood et al., 2003), where the composition of components into separate sub-systems supports minimal disruption during evolution.

5.3.3 Basic Update Evolution

A system may be evolved by processes other then decomposition and re-composition. For example, basic update can be used to change data values in an executing system. A program value in an updatable location can also be replaced by a new program. The advantage of decomposition over basic update is that evolutions can be applied to executing programs. However, the two process can be combined and a basic update can take place after decomposition as in the previous example.

5.3.4 Entity and Representation Domains

In terms of the entity (E) and representation (R) domains (Sec. 3.6.2), Hypercode graphs characterize E values in R, where they are manipulated by meta-programs. The R containing Hypercode graphs is not exactly the same as the R for Hypercode. In the standard R, values are reified to produce a visual representation for the programmer, in an environment where they may be edited, exploded and evaluated by hand. In the MPF representation domain (R_{MP}), values are reified to data structures that can be interpreted by a meta-program.

A fundamental difference between programmers and meta-programs is that whereas programmers operate exclusively in the R domain, meta-programs may access values in both R_{MP} and E. This is possible because the meta-programs themselves exist in E as opposed to programmers, who are outside it. For this reason, R_{MP} can be considered a sub-domain of E. The relationship between the three domains is illustrated in Figure 5.2. When reasoning about operations performed by meta-programs, such as evaluation, it is more intuitive to think of R_{MP} and E in the same relationship as the standard domains.

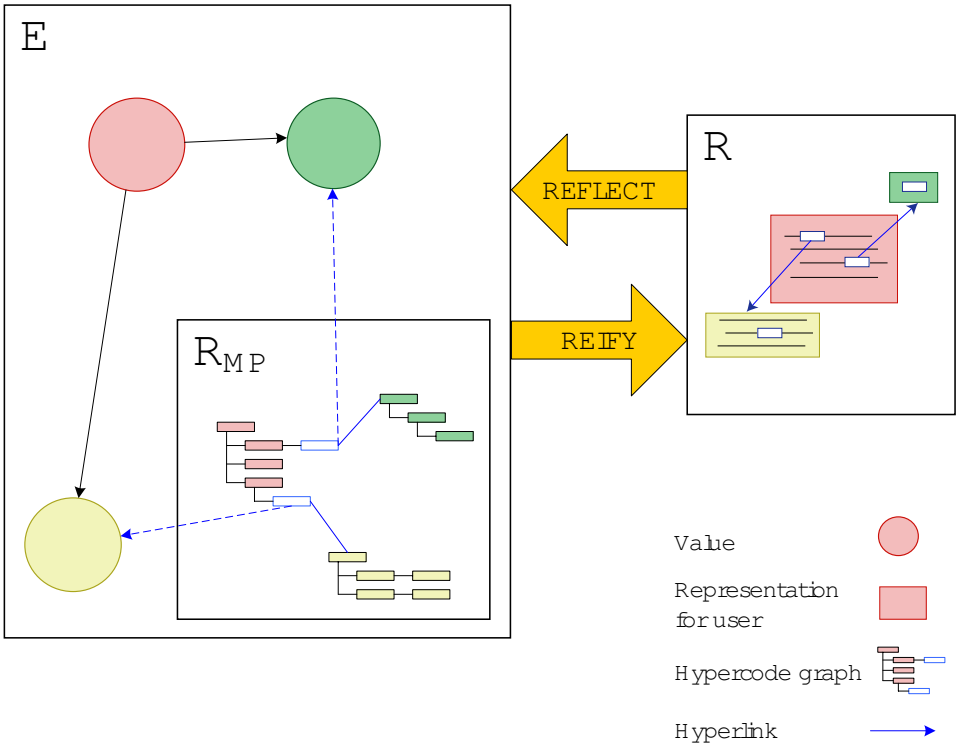


Figure 5.2. Domains. The R_{MP} domain exists inside the E domain. Meta-programs use hyperlinks to access both Hypercode graphs and the values they represent. In the R domain hyperlinks can only be used to access Hypercode representations.

Editing of a Hypercode graph takes place entirely in R_{MP} in the same way as editing of a visual Hypercode representation takes place in R. The evaluate and explode operations used by meta-programs have equivalent definitions in E and R_{MP} to those of the standard evaluate and explode in E and R. Meta-programs can, however, have direct access to values in the E domain. When traversing a Hypercode graph, a hyperlink may be dereferenced to give a value rather than the graph representation of that value that would be returned by explode. The Hypercode operations are shown in Figure 5.3 in terms of E and R_{MP}.

5.4 Evolution Patterns

Change management systems create evolution patterns for frequently used evolutions. They make the task of defining meta-programs easier by promoting re-use. Rainbow

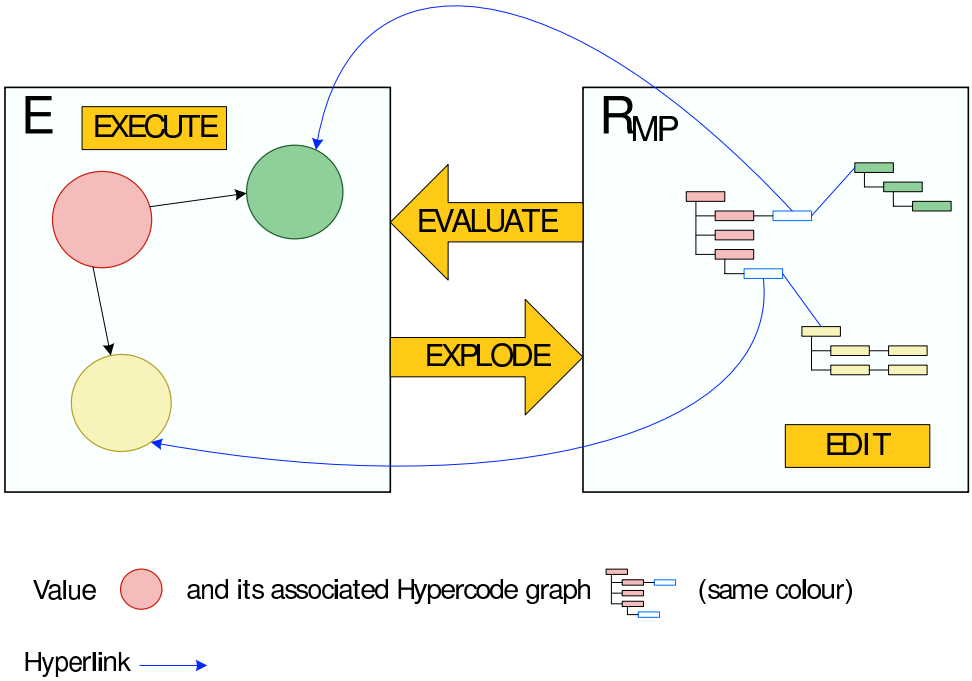


Figure 5.3. Domains Operations. Edit, evaluate, explode and execute can be described in terms of E and R_{MP}. However, hyperlinks in R_{MP} give direct access to values as well as their representations.

(Sec. 2.5.4) calls evolution patterns *architectural operators* and associates them with particular styles. For example, a style that uses services defines the *addService* and *removeService* operators.

As part of using the MPF for an incremental evolution process, an evolution pattern has been defined to recompose a set of decomposed components. It eases the task of a meta-program replacing one component in a composition with a new component. Without the pattern, a meta-program has to explicitly reconnect all the components as part of the re-composition. Assuming that the connections have not changed since before the decomposition, the pattern takes care of reconnecting them in the same way as before. This is just one example of an evolution pattern, in general they can be defined to do anything a meta-program can do, i.e., anything within the power of the programming language.

This evolution pattern is specific to the ArchWare ADL, where decomposing a component disconnects the connections between its sub-components. In order for the sub-components to continue communicating after the evolution they must be reconnected. The pattern deals with the case where one of the sub-components is replaced and its

replacement is a component that has the same connectivity. This new component may be an evolution of the sub-component it is taking the place of.

A sequence of MPF operations are used to define the evolution pattern. They are used in a given order at particular points in an incremental evolution. One set of operations is used at the decomposition stage to preserve information about the connections. Another set of operations is then used at the re-composition stage to reconnect the components in the same way using the information that was preserved. Table 5.1 defines the operations.

Table 5.1. Evolution Pattern Operations. The operations that make up an evolution pattern which replaces one sub-component of a decomposed component.

Name	Description
newDecomposeGraph (any) → graph	Create a graph representing the decompose clause for the given value, which must be a composed component.
decomposeGraph (graph) → decomposed graph, saved connections	Decompose a composed component, return the decompose sequence represented as a graph and a list of the connections which the decompose has disconnected. These two values are used by newComposeReplaceGraph to recompose a new system.
newComposeReplaceGraph (decomposed graph, saved connections, new graph, name) → graph	Construct a graph representing a compose statement that replaces one component of a composition with a new component.

Decomposition is broken down into two operations. The first is *newDecomposeGraph*, which takes a composed component and returns a graph representing a decompose statement for that value. The second operation is *decomposeGraph*, which evaluates the graph created by *newDecomposeGraph*. It returns a record containing the decomposed parts, as well as information about connections. This information is used in the re-composition to recreate the same connectivity as before the decomposition.

Re-composition is supported by an operation that constructs a graph to represent a composition and then evaluates it. In the graph, one sub-component of the previously decomposed component is replaced with a new component. The operation is *newComposeReplaceGraph*. It takes as parameters:

- The decompose sequence and connection information returned by the *decomposeGraph* operation.

- A Hypercode graph representing the new component.
- The name of the component that is being replaced.

It constructs and evaluates the composition, the result of which is a graph of a hyperlink referencing the recomposed component. *newComposeReplaceGraph* saves a meta-program having to extract the sub-components it is not updating from a decomposition sequence. It also saves the meta-program having to rewire the connections between components, a process which can otherwise be complicated in the ArchWare ADL.

5.5 Summary

The process of incremental evolution involves:

- Partially stopping an executing system using decomposition.
- Acquiring a Hypercode graph representation of the system to which changes are applied.
- Incorporating the changes into the executing system using structural reflection through the evaluate operation.

The evolution should proceed according to the Principles of Minimal Disruption and Unconfined Evolution.

Evolution patterns characterize common evolution processes in a change management framework. An evolution pattern for incremental evolution has been defined, which preserves information about a component's internal topology before decomposition. It then reconnects the sub-components in the same way when the component is recomposed after replacing one sub-component.

Chapter 6

Evolution Example

This chapter presents an example of using the Meta-Programming Framework to automate the evolution of an executing system according to the Principle of Minimal Disruption.

An evolution of an executing system using the MPF and following the Principle of Minimal Disruption is explained step by step. The example takes place in a Widget processing factory. One component in the factory's process produces Widgets, while another component consumes them. In the evolution, the producing component is updated to use a new function for Widget production. This involves replacing a component inside the producer, without interrupting the components executing in the consumer.

The steps taken by a meta-program to perform the evolution are as follows:

1. Start with a handle to a composed component.
2. Decompose the component to get a sequence of its sub-components.
3. Find the component in the sequence that is to be be updated.
4. Get a graph representation of it.
5. Alter the graph representation.
6. Evaluate the altered graph representation to get a new component value that has internal state from the old component.
7. Recompose the sequence of sub-components - replacing the updated one with the new component value.

The meta-program is an open adaptive engine, because it updates a component to use a function defined outside the system. The evolution is an illustration of component replacement (Sec. 1.5.3). It uses the evolution pattern defined in Section 5.4 to decompose, evolve and recompose a component.

6.1 Initial System Configuration

The initial system, depicted in Fig. 6.1, contains five components composed into two separate sub-systems. The first system contains two components: one produces *Widgets* and the other controls the operation. The second system contains a consumer and a controller. The components that operate on *Widgets* have shared state as they operate on objects in the same location, referenced from within the components by hyperlinks.

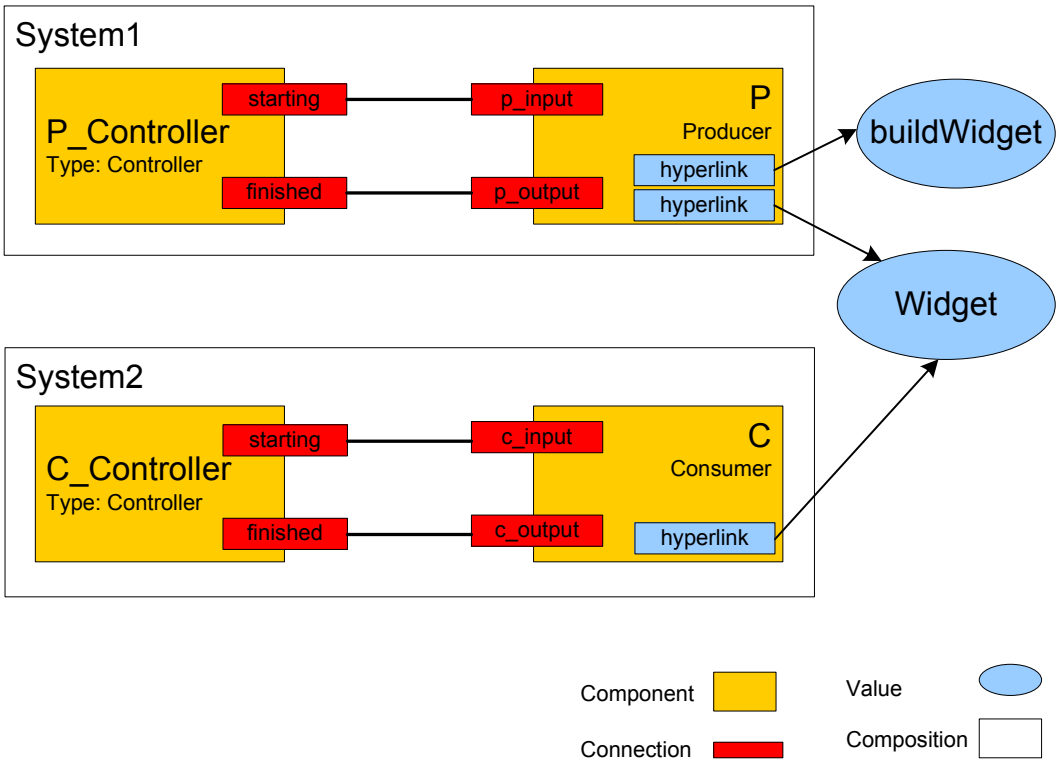


Figure 6.1. Initial System. The initial system where components operate on a Widget under the supervision of Controllers.

This system is a single producer / consumer pair example and does not provide a template for general producer / consumer systems or systems that can be evolved with the MPF.

System1 composes *P_Controller* (a *Controller*) and *P* (a *Producer*), which produces *Widgets*. *P_Controller* initiates *Widget* production by signalling to *P* on its *starting* connection. This causes *P* to produce a *Widget* before signalling to *P_Controller* that it has finished on its *p_output* connection. *P_Controller* then loops and the execution continues indefinitely. *System2* composes *C_Controller* (a *Controller*) and *C* (a *Consumer*), which consumes a *Widget* before signalling its completion to *C_Controller*.

System1 will be evolved by updating the *Producer* to use a new function to produce *Widgets*. Figure 6.2 shows the definitions of *Producer* and the component that is produced by the evolution, *Producer1*. The difference between them is that *Producer* uses

buildWidget to create a *Widget* whereas *Producer1* uses a newly defined *buildNewWidget* function. *Producer* and *Producer1* operate on a *Widget* in the same location because currentWidget, the hyperlink referencing it, has not been changed. This means that the new component operates on the same data as the component it is replacing.

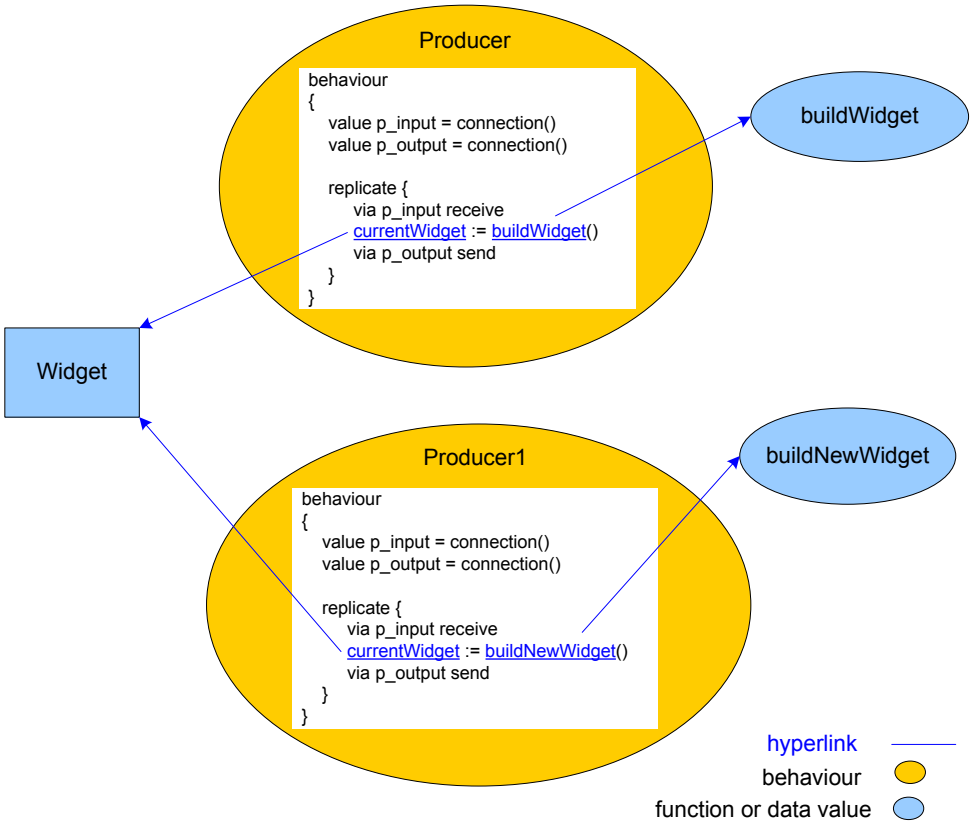


Figure 6.2. Updating the call to buildWidget. Producer and Producer1 differ in the function used to produce currentWidget.

The example uses two systems to demonstrate that one system can continue to execute while the other is decomposed and evolved. It also shows how an evolution pattern is used to disconnect and reconnect components (*P_Controller* and *P*.

6.1.1 Initial System Definition in ArchWare ADL

This evolution has been implemented using the MPF with ArchWare ADL. In the implementation, components are defined using ArchWare ADL *behaviours* (component

instances) and *abstractions* (parameterized component types). In Figure 6.1 the *Controllers* are defined as abstractions and the other components are behaviours. Abstractions are applied to create component instances.

Producer's definition is shown in Figure 6.3. It initially waits for a signal, an empty message, on the *p_input* connection (line 6). On receipt of the signal it produces a *Widget*, assigns it to the *currentWidget* (line 7) and signals that it is finished on the *p_output* connection (line 8). *Producer* will be evolved by replacing the call to *buildWidget* (line 7) with a call to a new function called *buildNewWidget*.

```
1  value Producer = behaviour
2  {
3      value p_input = connection() ! a connection for empty messages
4      value p_output = connection()

5      replicate {
6        via p_input receive ! receive start signal
7        currentWidget := buildWidget() ! produce a Widget
8        via p_output send   ! signal completion
9      }
10 }
```

Figure 6.3. Producer. The definition of Producer in ArchWare ADL.

currentWidget and the call to *buildWidget* (line 7) are both hyperlinks. The *currentWidget* hyperlinks in the *Producer* and the *Consumer* reference the same value, as was shown in Figure 6.1.

The *replicate* clause (line 5) corresponds to the π-calculus concept of replication and causes a new process (thread) to be started each time a signal is received on *p_input*. The new process executes the code inside the *replicate* clause.

Consumer is defined in Figure 6.4. It waits for a message on *c_input* (line 6) before calling the *currentWidget*'s *consume* function (line 7). When *Consumer* is finished it signals on the *c_output* connection (line 8). In *System2*'s composition, *c_output* will be unified with the *finished* connection in *C_Controller*.

The *Controller* (Fig. 6.5) abstraction manages a process by sending a start signal (line 6) and waiting for a finish signal (line 7) before starting the process again. A *while* loop (line 5) is used instead of a *replicate* clause because the first statement inside the loop

```
1  value Consumer = behaviour
2  {
3    value c_input = connection() ! a connection for empty messages
4    value c_output = connection()
5    replicate {
6      via c_input receive ! receive start signal
7      'currentWidget.consume()
8      via c_output send   ! signal completion
9    }
10 }
```

Figure 6.4. Consumer. The definition of Consumer in ArchWare ADL.

sends a message. A *replicate* clause must start with the receipt of a message to prevent infinitely many processes being spawned. *Controller* is applied in both *System1* and *System2* to keep the systems running indefinitely. Defining it as an abstraction allows two separate instances to be created.

```
1  value Controller = abstraction()
2  {
3    value starting = connection() ! a connection for empty messages
4    value finished = connection()
5    while true do {
6      via starting send       ! send start signal
7      via finished receive  ! receive finished signal
8    }
9    done
10 }
```

Figure 6.5. Controller. The definition of Controller in the ArchWare ADL.

System1 in Figure 6.6 composes *Producer*, labelled as *P* (line 2), and a *Controller* instance, labelled as *P_Controller* (line 3). As a result of the unifications (lines 5-6), *P_Controller* will send message to *P* to start it. On completion, *P* will signal back to *P_Controller*'s *finished* connection. *System1* is a mutable location (line 1) in order that it may be updated with an evolved version of the system after evolution. Figure 6.7 shows *System2*, which composes the *Consumer* (line 2) together with a *Controller* (line 3).

```
1  value System1 = location( compose {
2     P  as Producer  ! Producer labelled as P

      ! an instance of Controller labelled as P_Controller
3     and P_Controller  as Controller()

4     where {     ! unify the components' connections
5        P_Controller::starting unifies P::p_input,  ! P_Controller starts P

         ! P finishes and signals to P_Controller
6        P::p_output unifies P_Controller::finished
7     }
8  })
```

Figure 6.6. System1. System1 defined in ArchWare ADL. It composes Producer with a Controller.

```
1  value System2 = location( compose {
2     C  as Consumer     ! Consumer labelled as C

      ! an instance of Controller labelled as C_Controller
3     and C_Controller  as Controller()
4     where {     ! unify the components' connections
5        C_Controller::starting unifies C::c_input,   ! C_Controller start C

        ! C finishes and informs the C_Controller
6        C::c_output unifies C_Controller::finished
7     }
8  })
```

Figure 6.7. System2. System2 defined in ArchWare ADL. It composes Consumer and a Controller.

The system defined here is executing at the time the example starts. Code fragments in the following sections are from the meta-program performing the evolution.

6.2 Decompose

Decomposing the executing program breaks connections that were unified in the composition but does not stop the execution. This allows access to components which were encapsulated by the composition, as it returns a sequence of the composition's parts.

In combination with the program representation provided by the MPF, this means that starting with a single handle to a composed system, a value inside the system can be located and, if it is in a mutable location, updated. This evolution can occur without affecting parts of the system that are not communicating directly with components in the decomposed part.

The meta-program is an ArchWare ADL program in the same execution space as the example system. It starts with a handle to a mutable location containing *System1*, which it decomposes before locating *P* inside the decomposed system. Figure 6.8 shows the system after the decomposition, where the connections between *P_Controller* and *P* have been broken. The next section shows the code a meta-program uses to perform this decomposition.

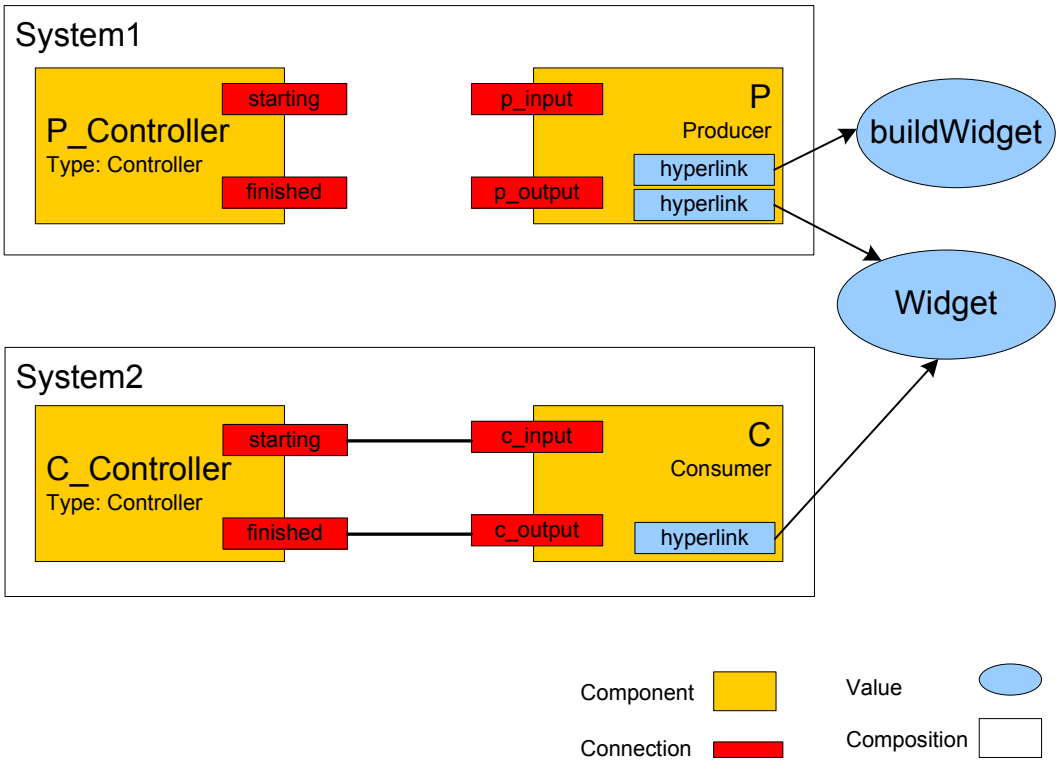


Figure 6.8. System1 Decomposed. The system after System1's decomposition. The difference from the initial configuration is that P_Controller and P are no longer connected.

6.2.1 Decompose System1

According to the evolution pattern, the first step in a decomposition is to use the *newDecomposeGraph* operation to construct a graph representing a program that decomposes the value. Passing *System1* as a parameter to *newDecomposeGraph* gives a Hypercode graph which, when evaluated, will decompose *System1*.

```
! Construct a graph representing the decomposition.
value decomposeGraph = newDecomposeGraph(System1)
```

The second step is to evaluate this graph using the *decomposeGraph* operation, which causes *System1* to be decomposed, as pictured in Figure 6.8. The operation preserves information about which connections were unified before the decomposition. This information is used later, when *System1* is recomposed after the evolution, enabling the connections to be re-unified automatically.

```
! Evaluate the decomposition graph.
value decomposeResult = decomposeGraph(decomposeGraph)
```

decomposeGraph returns a record with two parts: *saved_connections* and *decompose_sequence*. The former is used for re-composition in order that the components can be re-connected in the same way as they were before the decomposition. The latter is the result of the standard decompose operation and allows the meta-program access to the components in the composition. It is a sequence of two components, *P* and *P_Controller*, expressed as a graph (Fig. 6.9).

```
! Get the sequence returned by the decompose statement.
value decomposeSequence = decomposeResult.decompose_sequence
```

6.2.2 Locate P

The meta-program needs to know something about the composed system in order to locate the component to be replaced, e.g., the label of the component (in this case *P*) or its position in the sequence. In this example, the meta-program knows that *Producer* is the first component in *System1*'s composition.

To get a handle to *Producer*, the meta-program traverses the graph representing the sequence returned by decompose. The function *getFirstBehaviourFromDecompose* conveniently does the traversal, returning a hyperlink that references the component:

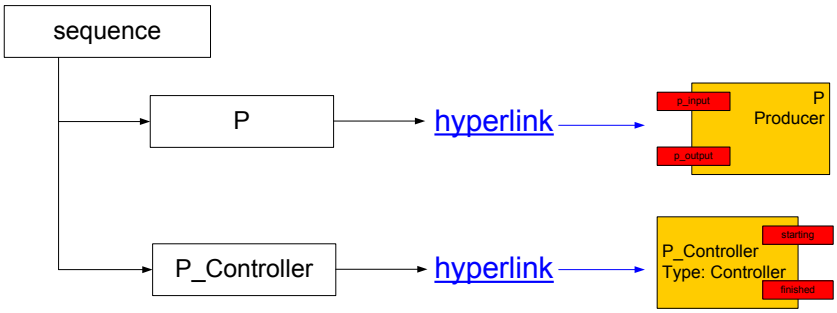


Figure 6.9. Decomposition Sequence. The decomposition sequence references the two sub-components in System1.

```
value ProducerHyperlink = getFirstBehaviourFromDecompose(decomposeSequence)
```

6.2.3 Get Graph Representation of Producer

At this point, the meta-program has a hyperlink to *Producer*. The *getGraphFromHyperlink* function is used to dereference the hyperlink and get a handle to a graph representation of *Producer*. This graph will be altered to produce a graph for *Producer1*.

```
value ProducerGraph = getGraphFromHyperlink(ProducerHyperlink)
```

6.3 Update Producer

Producer will be updated by changing the hyperlink that references *buildWidget* to a hyperlink that points to *buildNewWidget*. Figure 6.10 shows *Producer*'s graph and the hyperlink which will be changed. First, the function value that the new hyperlink will reference must be created. Then the meta-program will traverse the graph to find the hyperlink to be replaced.

6.3.1 Create buildNewWidget

To replace a hyperlink correctly, the meta-program needs to know the type of the value it references. Replacing a hyperlink with one referencing a value of a different type may cause a compilation error. A new function, *buildNewWidget*, with the same type

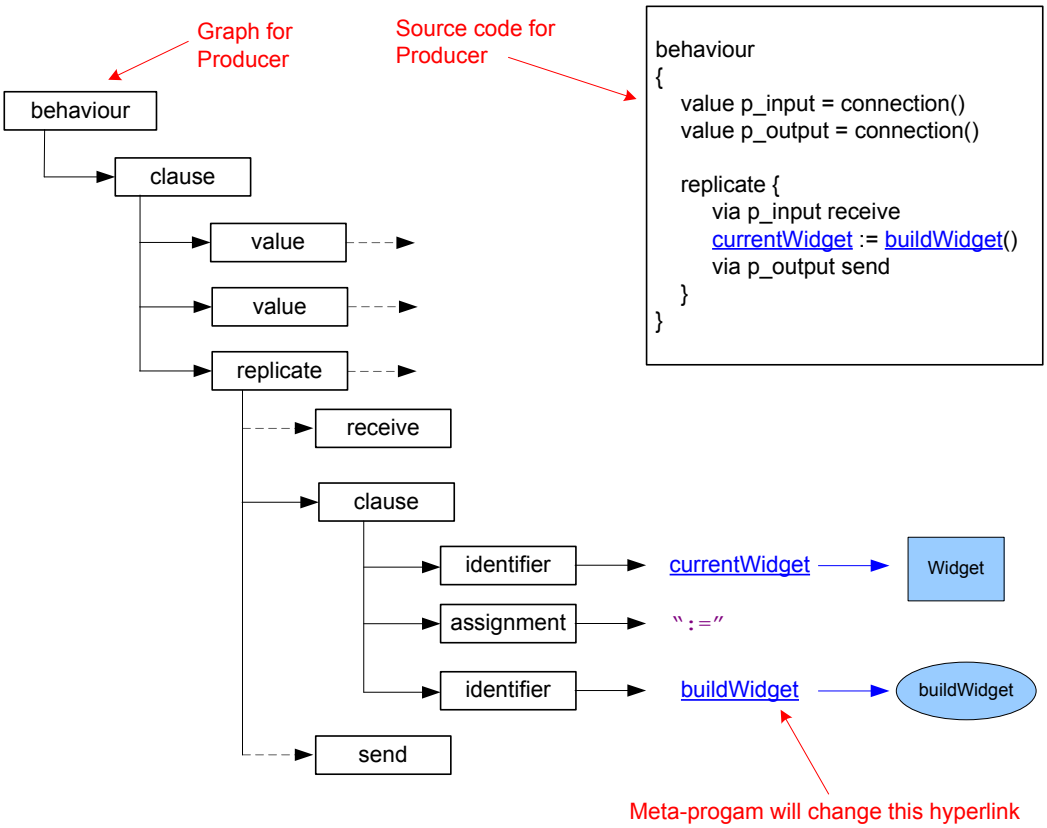


Figure 6.10. Producer Graph. The meta-program has the graph representation of Producer, from which it will make Producer1 by changing the hyperlink.

will replace *buildWidget*. The meta-program can find out the type of *buildWidget* by parsing the type information available in its hyperlink node.

buildNewWidget is defined using a string of ArchWare ADL supplied by a developer, whose intervention is necessary to introduce new functionality. The *compile* function (Sec. 3.7), which is part of the standard function set available in the ArchWare ADL is used to reflect the string representation of *buildNewWidget* into an ArchWare ADL value. It takes a *string* representing the code to be compiled and returns the compiled value.

The following process is used to create the new function and a hyperlink referencing it:

1. Define *buildNewWidget* as a string of ArchWare ADL.

```
value buildNewWidgetString = " .... "
```

2. Compile the buildNewWidget function.

```
value compilationResult = compile(buildNewWidgetString)
```

3. The result of compilation is a function, typed as an infinite union type, that must be executed to get a handle to *buildNewWidget*. Before it can be executed it must be projected onto a function type.

```
1  value buildNewWidget = project compilationResult.result as X onto
2    function[]→any: X() ! The result of X() is the buildNewWidget function.
3    default: any("")
```

The result of compilation is projected onto a function type in line 2 and then executed. If the compilation was not successful the default clause in line 3 will be executed instead. The meta-program now has a handle to *buildNewWidget*.

4. A hyperlink that references *buildNewWidget* is created and associated with *Producer*'s graph using the *newHLElement* operation. Hyperlinks need to be associated with a graph to add the values they reference to the closure of the program the graph represents.

```
value buildNewWidgetHyperlink =
                    newHLElement(buildNewWidget,"buildNewWidget",Producer)
```

6.3.2 Replace Hyperlink

Producer's Hypercode graph is updated by replacing the hyperlink to *buildWidget* with the new hyperlink to *buildNewWidget*. The result is a Hypercode graph representing *Producer1*.

The meta-program uses traversal operations on *Producer*'s Hypercode graph to find the hyperlink that will be replaced. *getAllHyperlinks* returns a list of the hyperlinks in *Producer*'s graph, providing an easy way for the meta-program to find the *buildWidget* hyperlink without knowing anything about the structure of the rest of the graph.

```
value hyperlinkList = getAllHyperlinks(ProducerGraph)
```

The following code in the meta-program searches through the list of hyperlinks until one is found called *buildWidget*:

```
value list =  location(hyperlinkList)   ! variable to store the search result

! search the list until a hyperlink with the name buildWidget is found
while 'list.node.name  <> "buildWidget"  do list := '('list.next)
value buildWidgetHyperlink = 'list.node  ! assign search result
```

The *replaceChild* function is used to replace the *buildWidget* hyperlink with the newly created *buildNewWidget* hyperlink.

```
replaceChild(buildWidgetHyperlink,buildNewWidgetHyperlink)
```

Apart from the hyperlink that has been replaced, the hyperlinks in *Producer* have been preserved over these changes. When the graph is evaluated a new component is created containing hyperlinks referencing the same values as in the original component. For example, currentWidget, seen as a hyperlink in Figure 6.3, still refers to the same extant value after the evaluation.

At this stage, *Producer1* exists in the form of *Producer*'s altered graph representation. The graph's evaluation returns a hyperlink referencing the new value *Producer1*. Before evaluation, the operation *newDeclarationGraphForGraph* is called to extend the graph to be a definition of a new value with the name *Producer1*.

```
value Producer1Graph = newDeclarationGraphForGraph("Producer1",ProducerGraph)
```

The *evaluateGraph* operation performs the evaluation.

```
value Producer1 = evaluateGraph(Producer1Graph)
```

6.4 Recompose System1

The final part of the evolution pattern involves re-composing *System1* using *Producer1* and the original *P_Controller* component. As described in Chapter 5, in the evolution pattern, recompose is distinguished from a standard composition because it uses information about connections that was saved during decomposition. Connections that were unified in the original compose statement are re-unified.

The parameters for the *newComposeReplaceGraph* operation used to recompose are:

decomposeResult.decompose_sequence The sequence obtained by decomposing *System1* (Fig. 6.9).

decomposeResult.saved_connections The set of connections between components as they were before decomposition.

Producer1 The new component.

P The label of the component being replaced. This label was attributed to *Producer* in the initial composition shown in Figure 6.1.

```
value composeGraph =
    newComposeReplaceGraph(decomposeResult.decompose_sequence,
                           decompose_sequence.saved_connections,
                           Producer1, "P")
```

The result of *newComposeReplaceGraph* is a graph representing a compose statement for *System1a*. Figure 6.11 shows its components and connections, as well as the source code for the compose statement.

The compose graph is evaluated as follows:

```
value System1a = evaluateGraph(composeGraph)
```

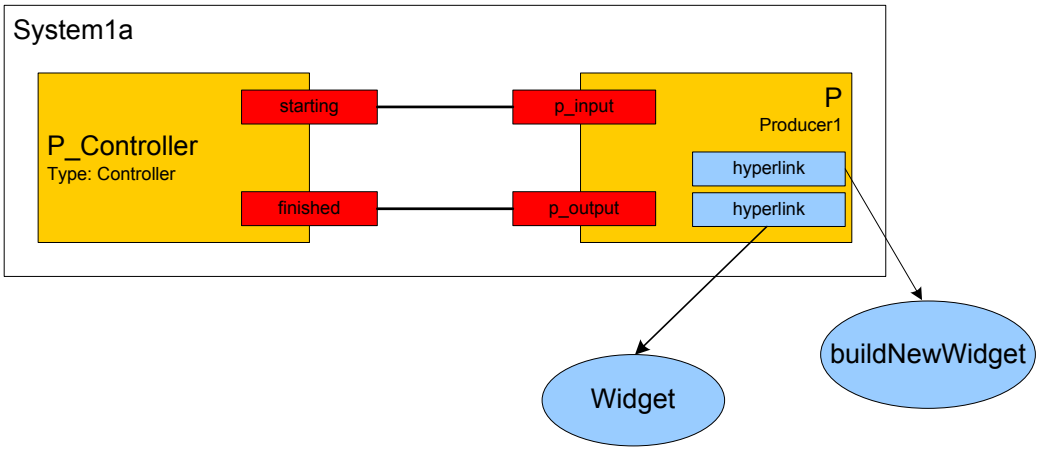
It remains for the meta-program to update the location that contained *System1* using the *updateHyperlinkLocation* operation:

```
updateHyperlinkLocation(System1,evolvedSystem)
```

The meta-program is complete and the system has been evolved. State has been preserved by maintaining hyperlinks over the change (currentWidget in *Producer*) and the rest of the system continued to execute during the evolution. The use of MPF operations and an evolution pattern means that, if some program values were parameterized, the meta-program could be used to update another system with the same structure.

6.5 Summary

This chapter gives an example of evolution where one component in an executing system is updated by a meta-program. The example follows the Principle of Minimal




```
compose {
   P as Producer1
   and P_Controller as Controller()

   where {
      P_Controller::starting unifies P::p_input,
      P::p_output unifies P_Controller::finished
   }
}
```

Figure 6.11. System1a. System1a composes Controller (unchanged from System1) and Producer1. The Hypercode graph produced by newComposeReplaceGraph represents the code for the compose statement shown here.

Disruption as other components continue to execute during the evolution and extant state in the updated component is preserved. The code for the example is in Appendix C.

The steps taken by the meta-program are:

1. Start with a handle to a composition of components.
2. Decompose to get the sequence of sub-components.
3. Find the component in the sequence that will be updated.
4. Get a graph representation of it.
5. Alter the graph representation.

6. Evaluate the altered graph representation to get a new component value.

7. Recompose the set of sub-components - replacing one with the new component value.

The diagram in Figure 6.12 illustrates this process. A component is decomposed (1) to get a sequence of sub-components. Dereferencing a hyperlink in the sequence gives a Hypercode graph representing the sub-component (2). The graph is updated (3) and evaluated (4) to create a new value. The component is then recomposed (5) to include the new sub-component.

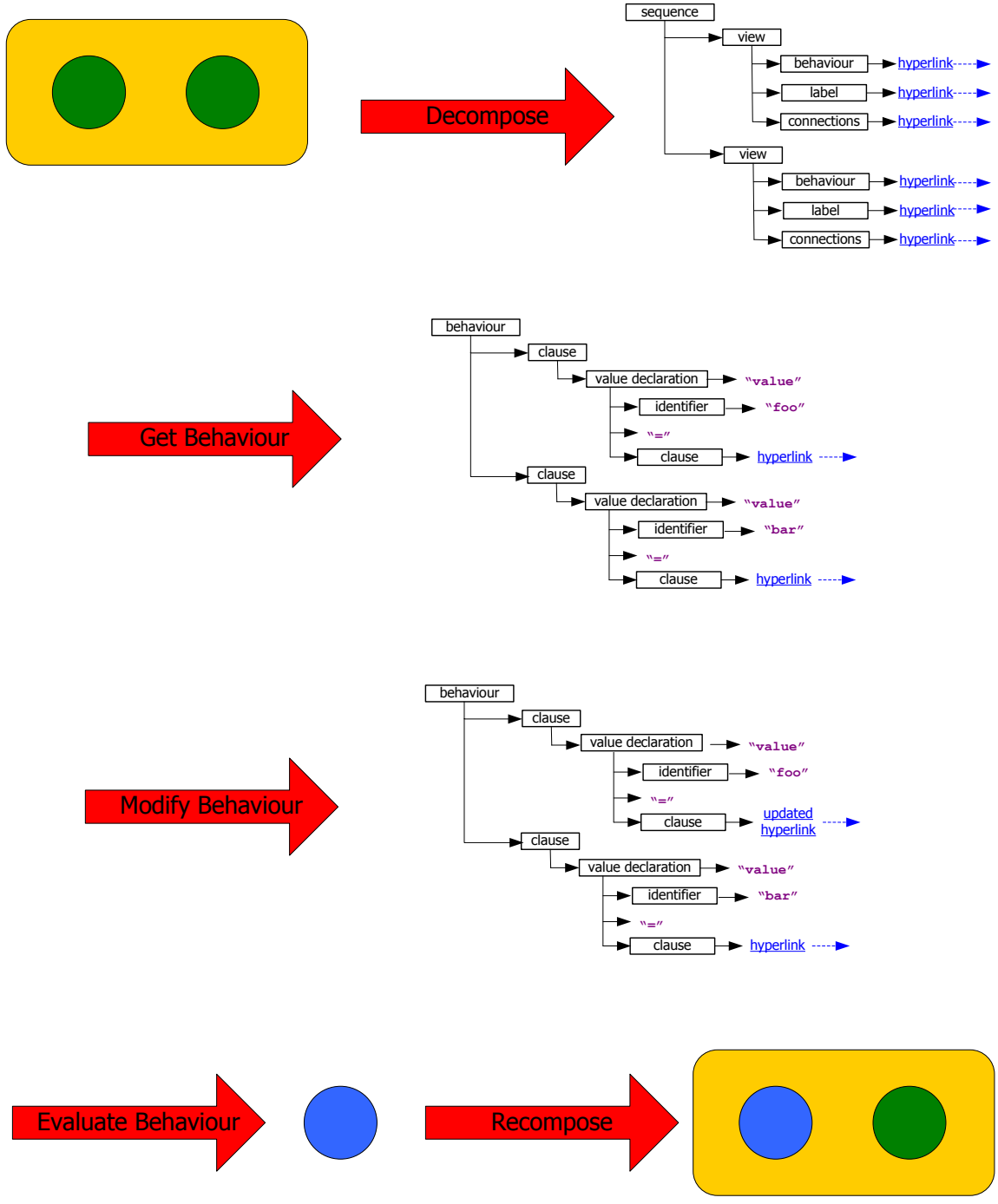


Figure 6.12. Evolution. The steps taken by a meta-program updating a component in a composition.

Chapter 7

Implementation

This chapter details the implementation of the Meta-Programming Framework including the data structures used for Hypercode graphs, the use of generative programming techniques to evaluate and explode Hypercode graphs, and the MPF operations.

The MPF defines Hypercode graphs and an API that can be used to evolve executing systems. Hypercode graphs represent executing programs and implementing them requires a means of representing every data type in the universe of discourse, including: scalars, user defined types and reference types. The Hypercode operations evaluate and explode allow Hypercode graphs to be executed and introspected. The API, an interface for meta-programs to interact with Hypercode graphs, is defined by the MPF operations.

A summary of the overall structure is followed by a specification of the data structures used to represent Hypercode graphs. The use of generators to realize the evaluate and explode operations is explained and the MPF operations are described. Finally, an interface that allows users to interact with Hypercode graphs using the same mechanisms as meta-programs is presented.

7.1 System Structure

Figure 7.1 gives an overview of the framework's software architecture, where each layer uses mechanisms provided by the layer below. It shows that a meta-program uses operations provided by the MPF, which in turn uses the functionality of a Hypercode system. The Hypercode system includes *Hypercode Representations* and uses the ArchWare ADL compiler.

Meta-programs interact with Hypercode graphs using the MPF operations. These operations rely on the Hypercode operations *evaluate* and *explode*, which use data structures called Hypercode Representations to model the graphs. The ArchWare ADL compiler is used as part of the evaluation process.

In addition to the MPF's interface for meta-programs, there is a user interface, which provides the Hypercode operations *edit* and *implode*. A user may view, edit and evolve Hypercode programs through the interface.

7.2 Hypercode Graphs and Hypercode Representations

Hypercode graphs represent programs by augmenting program syntax with hyperlinks to the data graph. Within the Hypercode system, Hypercode graphs are implemented using data structures called Hypercode Representations (HCRs).

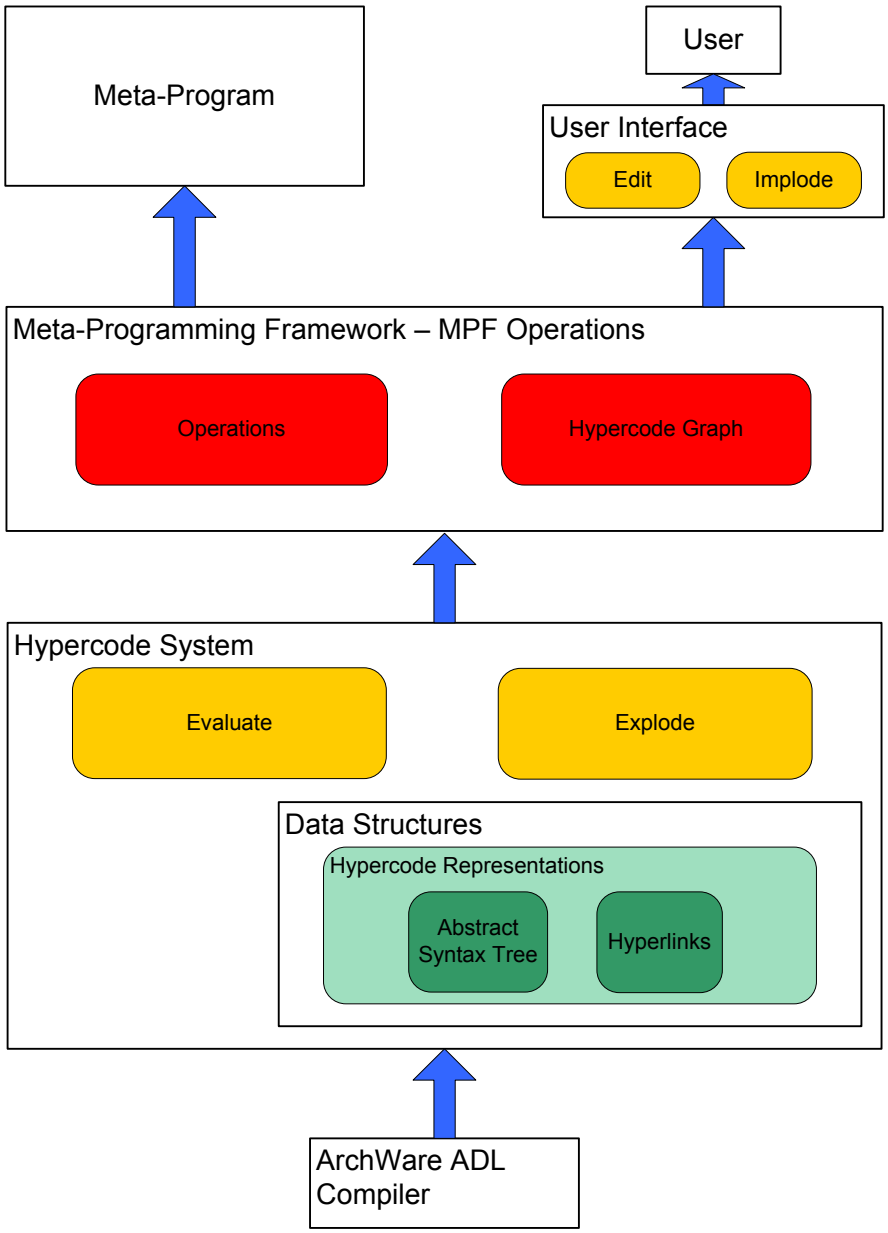


Figure 7.1. Architecture Overview. The components in the Framework implementation and their main functions. Each layer uses the mechanisms provided by the layer below.

Fine grained manipulation of Hypercode graphs is supported to allow unrestricted changes by meta-programs. Hypercode graphs and values are associated with each other and both available to meta-programs so that a meta-program dereferencing a hyperlink can access either a Hypercode graph representation or the value itself. The implementation is reusable in that the Hypercode graphs and Hypercode operations can be re-used for different programming languages.

In a Hypercode graph implementation:

- Graphs must be able to be generated at runtime to show the current state of variables and to allow for the representation of user defined types.
- Hyperlinks are used to model reference types. Multiple identifiers in a program's source code that represent the same value should become hyperlinks that reference the same value.
- Program source and closure must be available to support the generation of Hypercode graphs for code values.

7.2.1 Hypercode Graphs

The type of a Hypercode graph was described in Chapter 4 and the definition is repeated here in Figure 7.2. Graphs have three types of nodes: program syntax, text and hyperlink. An illustration of a Hypercode graph is shown in Figure 7.3 (repeated from Chapter 4). It shows how hyperlinks from the abstract syntax tree allow traversal of the full program closure.

Meta-programs can access a Hypercode graph for any first class value. Graphs for data values are generated by the Hypercode system when they are accessed by a meta-program. Graphs for code values are created at compilation time and stored in their closure. A compilation time syntax parse returns the source code marked up[1] with syntax information (Sec. 7.2.1), which is used to generate the graph. Identifiers in the code are turned into hyperlinks at the same time. The parsing could take place outside the compiler, however, because a full syntax analysis is required to generate the abstract syntax tree, it is most efficient to include it in the syntax analysis performed at compilation time.

[1] Mark-up languages, e.g., TeX and XML, combine text with extra information about the text.

```
type graph is record
        (
          node: nodeType,
          parent: graph,
          children: list of type graph
        )

type nodeType is record
        (
          name: string,
          text: string,
          attributes: list of (name, value) pairs
        )
```

Figure 7.2. Hypercode Graph Definition. The type definition of a Hypercode graph in pseudocode.

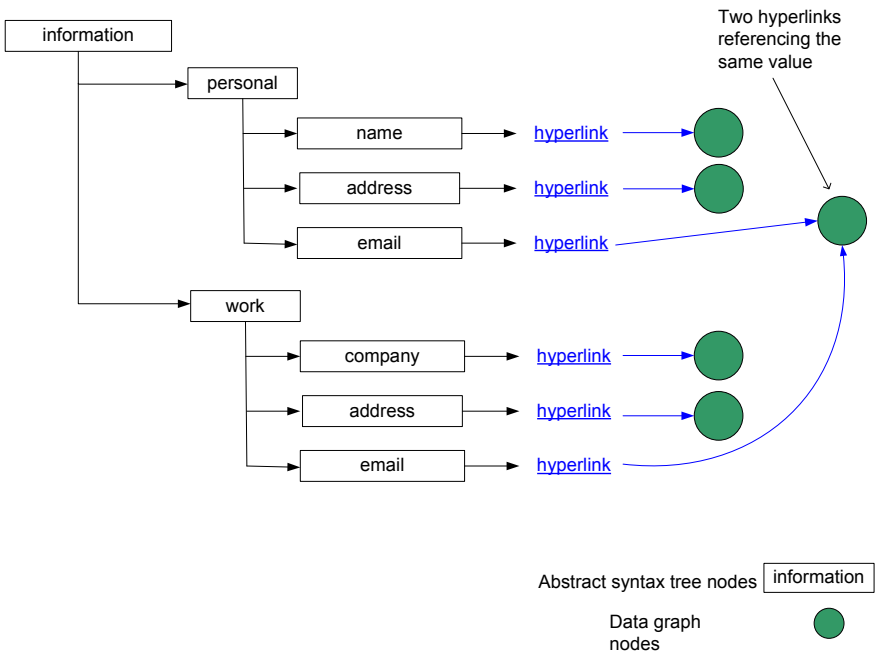


Figure 7.3. Hypercode Graph. A Hypercode Graph incorporates program syntax and existing values.

Hyperlinks

Hyperlink nodes reference values with a level of indirection. The value of an ID attribute (shown in Fig. 7.4) is used to locate a hyperlink in the list of hyperlinks that is part of an HCR data structure. Using indirect rather than direct references to values allows the Hypercode graph to remain valid outside the domain of the executing system. This permitted the construction of a visualizer and user-driven development interface for Hypercode (Sec. 7.5). The user interface is written in Java but can operate with Hypercode graphs in any language because the graph does not contain direct references to, for example, ArchWare ADL values.

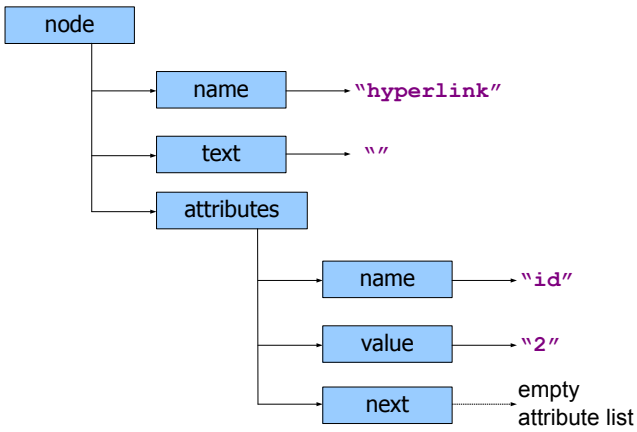


Figure 7.4. Hyperlink Node. A hyperlink node with an ID value of 2.

The IDs of hyperlinks are valid within the context of their HCR. To copy a hyperlink from one graph to another, a meta-program uses the *copyHyperlink* and *pasteHyperlink* operations. *copyHyperlink* creates a graph that is a hyperlink node contained in its HCR. *pasteHyperlink* uses the ID information in the graph from *copyHyperlink* to dereference the hyperlink. It then adds the obtained value to the list of hyperlinks associated with the new graph into which the hyperlink is being pasted. Finally, it inserts the hyperlink as a child of the graph node. These operations ensure that the hyperlink is valid within the context of the graph into which it has been pasted.

XML

The abstract syntax tree and hyperlink structure of a Hypercode graph is stored in the HCR using XML. Advantages of the approach include:

- XML is a convenient way of representing a tree.
- XML representations are easily viewable making development and debugging simpler.
- The standard format allows use of standard XML parsers.
- Hypercode graphs can be re-used by other programs, e.g., the user interface, that can easily import the XML format.
- In order to accommodate outside interactions and open evolutions, a meta-program may read in predefined Hypercode programs from a file to introduce them into the system. The XML format makes this process straightforward.
- The existing graph representation may be extended by the addition of new XML tags, e.g., to define new abstractions such as software architecture properties.

Extending the context free syntax (defined in Appendix A.1) requires only the addition of the relevant new XML tags. These must be incorporated in the syntax parser, which generates an XML representation from source code. Adding new tags to the definition should not affect the XML parser, which produces a Hypercode graph from the XML document, as it relies only on the structure provided by start and end tags and does not use information in the tag names. Operations over the abstract syntax tree remain unaffected because they ignore the tags they do not directly use.

The XML representation can be re-used for different programming languages. The program source code and some of the tags change but the Hypercode graph structure remains unchanged.

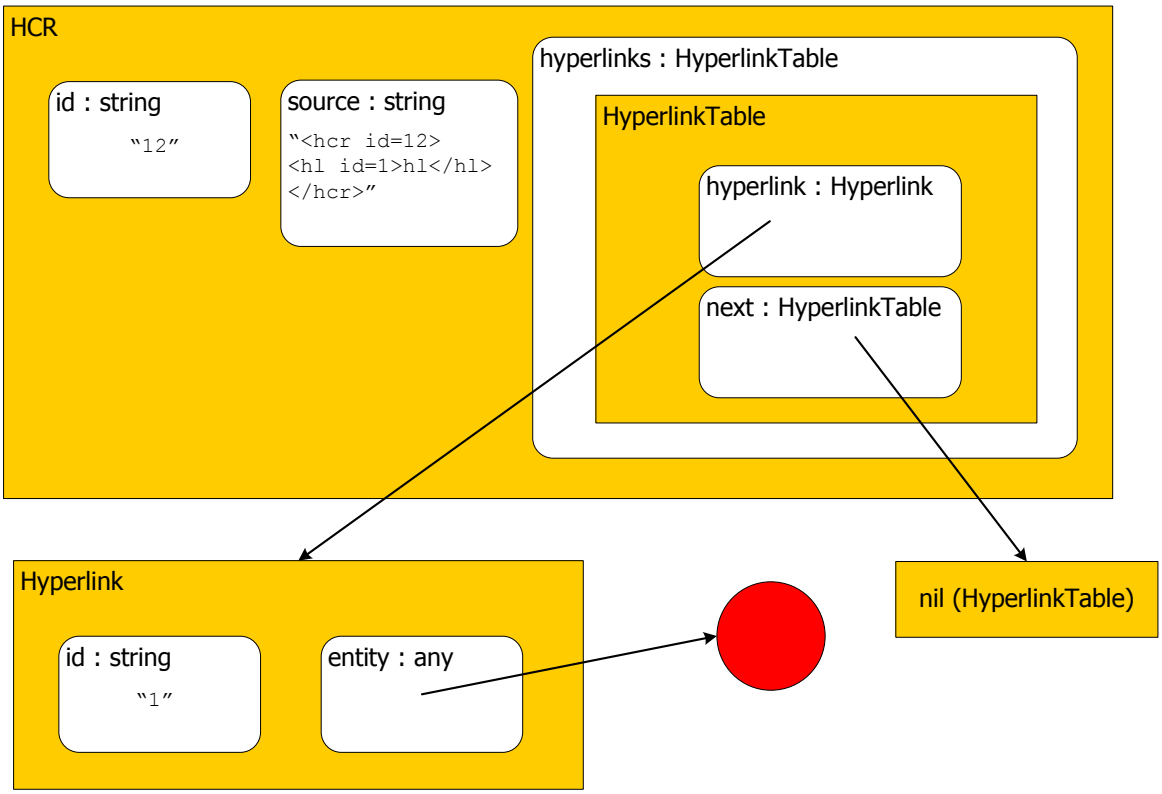
7.2.2 Hypercode Representations

Hypercode Representations (HCRs) are records containing a field for source code, typed as a string of XML, and a field for a list of hyperlink values. References from the source code to the list of hyperlinks are hyperlink tags.

An illustration of the elements in a HCR is shown in Figure 7.5 along with a type definition in pseudocode. Both HCRs and Hyperlinks have an *id*, a unique identifier, and are typed as records. HCR IDs identify HCRs in a table of those in current use. This table is maintained by the Hypercode system, which is session based. This is necessary

because Hypercode graphs use IDs as indirect pointers to values. The values themselves are not included in the graphs. As long as a meta-program is using a Hypercode graph the Hypercode system needs to keep the HCR data structure so that it can dereference the hyperlinks. Sessions are needed to allow the HCRs that accumulate in the Hypercode system to be garbage collected.

The source code of an HCR is stored as a string of XML and its hyperlinks are in a *HyperlinkTable*, which is a linked list. Hyperlink IDs are used to locate hyperlinks in the *HyperlinkTable*. Hyperlinks contain their ID and a reference to an extant value.




```
type Hyperlink is record ( id: string, entity: any )
type HyperlinkTable is list ( hyperlink: Hyperlink, next: HyperlinkTable )
type HCR is record ( id: string, source: string, hyperlinks: HyperlinkTable )
```

Figure 7.5. Hyperlink and HCR Type Definition. The type definitions for Hyperlinks and HCRs graphically and in pseudocode.

7.2.3 Functions

The HCRs for code values are generated at compilation time and stored in the object closure of the value that is created. Source code must be retained at compilation time since original program source code cannot be generated from a code value. (Reverse engineering may be able to generate program source code but it will not necessarily be the same as the original and it is faster and simpler to retain it.)

In the implementation of Hypercode for ArchWare ADL, there are three types that require source code retention: *abstractions*, *behaviours* and *functions*. All three will be referred to as code values. The first task in creating an HCR is generating an abstract syntax tree from the source code. The second is replacing identifiers with hyperlinks, which involves generating code for hyperlinks referencing values that may not exist at compile time. These are values that will be created during the execution of the code to be compiled, as opposed to hyperlinks that are already in the code before compilation and reference extant values.

Generate XML from Original Source

The abstract syntax tree is generated as XML mark-up in the program source code. To generate the XML, functions in the compiler are modified to output XML as well as object code. After the modification, the compiler generates a separate string of XML. This string contains all the code that the compiler has read in, interspersed with XML tags containing syntax information. For example, the function which processes strings outputs: `<string> processed string </string>`. After the compiler parses a code value, i.e. after generating the object code containing the code value definition, it outputs object code in which a new HCR is created and the string of XML assigned to its *source* field. When the code is executed, the HCR is stored in the closure of the code value as shown in Figure 7.6.

This process is clear cut when considering the complete original source code, which is represented in a single HCR. However, it becomes more complicated when taking into account multiple nested code values. Each of these code values must have its own HCR containing the relevant portion of the source code. The multiple, nested portions of source code are handled by maintaining a data structure with all the source code for each scope. The source code for a particular code value is the source code stored for the relevant scope level. Code is only stored for one code value at a time in the current

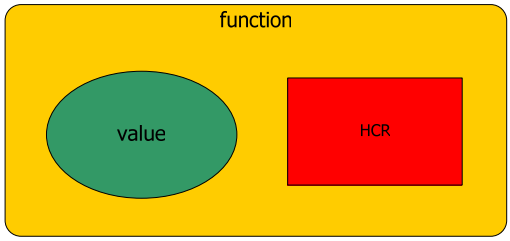


Figure 7.6. Code values with HCRs. Code values, e.g., functions, have an HCR in their closure.

scope level. At the end of a code value definition, its code is discarded at that level. The code may still be retained as part of a definition in an outer scope.

Hyperlinks in Code Values

The compiler considers hyperlinks in two categories. The first category comprises hyperlinks extant in the original source code. These are processed by including their hyperlink tags and IDs in the XML output string. The second category contains hyperlinks created in the compilation process.

The compilation process produces an HCR for each code value in the program and each one contains only that part of the source code that is in the scope of the code. For example, the code here could be a function definition within a larger program. The function definition may contain references to identifiers that are declared in an outer scope. The HCR for this function contains only its own code and when viewed in isolation, the identifiers declared elsewhere are undefined within the function. Therefore, in order that the HCR for the nested code value (i.e. the function) contains a correct program, identifiers declared in the outer scope are turned into hyperlinks. Figure 7.7 shows a function nested in a behaviour definition. The HCR for the function has hyperlinks in its source referencing *counter*, a variable declared in an outer scope.

The same problem, i.e., retaining references to variables declared in an outer scope, is faced when writing compilers for languages with first class procedures. A first class procedure can execute outside the domain in which it was declared. Therefore, the procedure's environment must be accessible from the procedure. Either a single pointer (static link) to the environment, or a pointer for each individual variable value can be used. The latter approach has been taken for the Hypercode implementation. The

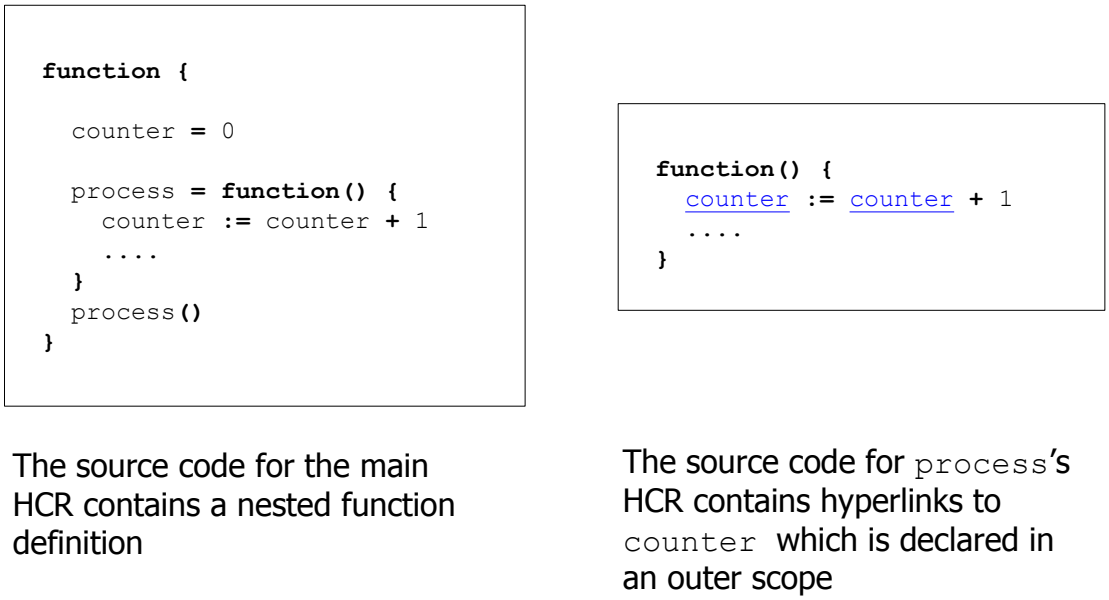


Figure 7.7. Hyperlinks in nested code values. An example showing that identifiers declared in an outer scope of a nested function become hyperlinks. The source code is shown as in a user interface.

same technique was used in the implementation of the PamCase Machine (Cutts et al., 2000), an abstract machine for a persistent system.

Creating a hyperlink in the compiler requires firstly, a hyperlink tag to be output into the generated XML, and secondly, a reference to the value to be added to the HCR's list of hyperlinks. A hyperlink tag is generated by creating a unique ID. Creating a reference to the value presents more of a problem because at compilation time the value does not exist. Therefore, the compiler generates object code to create the reference after the value is created. It uses the following algorithm:

1. If the declaration of the current identifier was in an outer scope:

   (a) Generate code that creates a reference to the value and adds the reference to the list of hyperlinks in the HCR.

(b) Concatenate the generated code to the code containing the definitions of other hyperlinks in the code.

(c) If the end of the code value's scope is reached, output the generated code into the code stream being created by the compiler.

Instead of just identifiers declared in an outer scope, all identifiers in the original source code may become hyperlinks, but this is not a requirement for a valid Hypercode representation. Identifiers declared in an outer scope are undefined in the code value, which is therefore not a valid program in its own right unless it includes hyperlinks to such values. On contrast, identifiers declared within the code value are defined and do not need to be hyperlinked for the code to be valid.

7.2.4 Data Values

The Hypercode graph of a data value is generated every time the value is accessed by a meta-program using the *getGraphFromHyperlink* operation. This guaranteeing that the graphs of mutable values are always up to date. For example, consider the hyperlink to *counter* in Figure 7.7. The meta-program will get a correct value for the counter every time it accesses the value through the hyperlink even though the value of *counter* may change. The explode operation, explained in Section 7.3.2, is used to generate the graphs of data values.

7.3 Implementing Hypercode Operations using Generators

The Hypercode system has been implemented using generative technology (Czarnecki and Eisenecker, 2000). Generators are used to transform a Hypercode graph into an equivalent program without hyperlinks that can be compiled with a standard compiler. Transformations are applied to the Hypercode graph according to a set of rules. This allows the construction of a Hypercode system that can be applied to different programming languages by changing the output of the generator's transformations. The generative programming technique used is *source-to-source transformation*. It allows the Hypercode system to use existing compilers and have a small core that only contains the generators instead of a complete Hypercode compiler.

The generic design of the Hypercode system means it can be applied to different programming languages with relative ease. Hypercode graphs can model values in any language and the graphs can be evaluated by plugging a small set of rules into the generator and then using a standard compiler. The explode operation, which generates Hypercode graphs for data values, uses another set of rules that can be plugged in.

Both the evaluate and explode operations were originally constructed for a ProcessBase Hypercode system. Minimal changes were necessary to convert the implementation to ArchWare ADL Hypercode. The system operates on the Hypercode graph structure and the programming language is largely irrelevant. Changes included use of an ArchWare ADL compiler instead of the ProcessBase compiler, and updating some generators to produce ArchWare ADL code equivalent to the ProcessBase code they were previously outputting. The examples in this section are in pseudocode.

7.3.1 Evaluate: Using Generators to Compile and Execute Hypercode

The evaluate operation compiles and executes a Hypercode graph and returns a Hypercode graph representation of the result of execution. The challenge in implementing evaluate lies in linking existing values, referenced by hyperlinks in the Hypercode graph, into the executable code. Part of this work has been published in the ACSC paper: Using Generative Programming to Visualise Hypercode in Complex and Dynamic Systems (Mickan et al., 2004).

Evaluate operates on a Hypercode graph in four steps: transformation, compilation, binding and execution. The progression, shown in Figure 7.8, produces a Hypercode graph representation. Generators map the original Hypercode onto some target code according to a set of rules. Compiling, binding and executing the target code completes the evaluation process.

Figure 7.9 shows how evaluate can be defined in terms of the functions applied to the Hypercode, where h is some Hypercode.

A Hypercode graph can be flattened to an XML string which is the HCR source. However, the graph is still a mixture of source code and hyperlinks and are therefore not suitable for compilation by a standard compiler, which operates on purely textual representations and is not able to incorporate the extant values referenced by hyperlinks

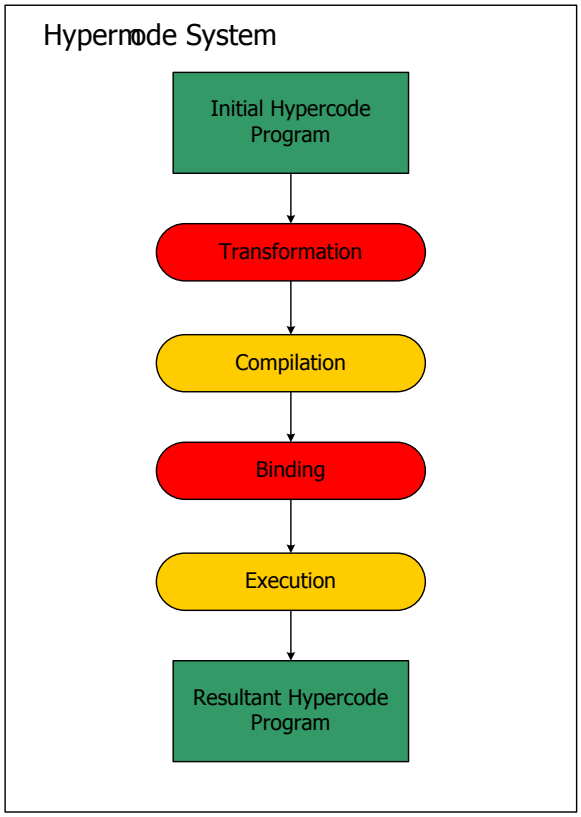


Figure 7.8. evaluate. The steps of the evaluation process.

evaluate (h) = explode (execute (compile (transform (h))))

Figure 7.9. evaluate. Applying evaluate to a Hypercode representation

into its output. The generators produce a program equivalent to the Hypercode graph, which can be compiled because it contains only text and no hyperlinks.

Reducing a Hypercode graph to a textual representation causes the references to extant values to be lost. The transformed code may be compilable but it is not equivalent to the Hypercode graph unless the extant values are bound into it. Therefore, these values are bound in after compilation using a *generator* function, which takes the values as parameters. Executing *generator* produces an executable and complete version of the original Hypercode graph. The two steps shown in red in Figure 7.8, transformation and binding, will be described in the following sections.

An overview of the steps of evaluate is depicted in Figure 7.10. The input is a Hypercode graph representation including hyperlinks.

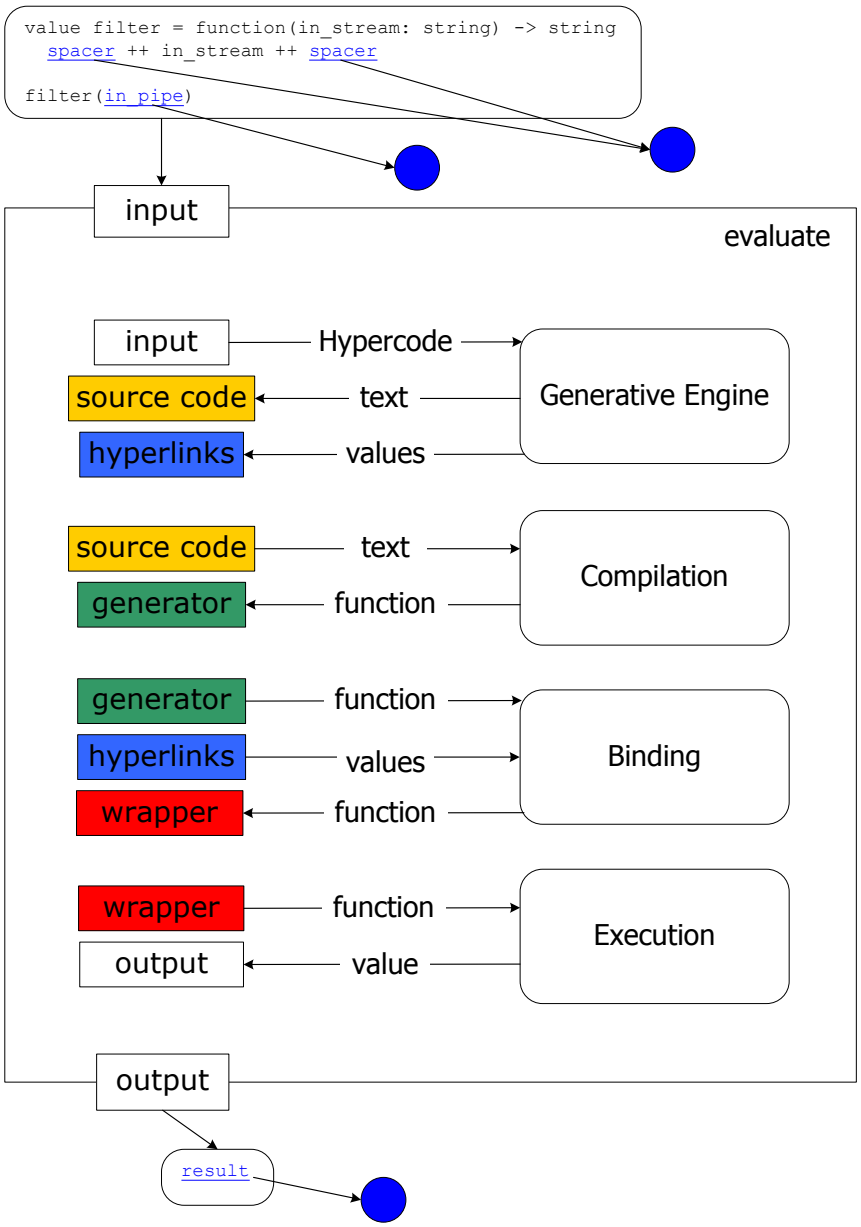


Figure 7.10. evaluate. The steps of the evaluate operation

The following steps define the algorithm for evaluation:

1. The Hypercode graph is input to the generative engine, which outputs the generated source code and a list of hyperlink values.

2. The source code is then fed to the compiler which produces the function executable *generator*.

3. The binding step involves executing *generator*, which takes the hyperlink values as parameters. *Generator* returns a function called *wrapper* that wraps the original source code and in which the hyperlink values are bound.

4. The final step is executing *wrapper* and returning the resultant value as a Hypercode graph.

Transforming Hyperlinks

The example code in Figure 7.11 will be used to show how the transformation progresses in the generative engine. It defines a function *filter*, which takes a string as a parameter and returns that string concatenated with two *spacers*. The spacer hyperlinks both reference the same value. The *filter* function is called with the hyperlink in_pipe as its input parameter. This produces a value which, expressed as a Hypercode graph, is the final result of the evaluation process.

```
filter = function(in_stream: string) → string
  spacer ++ in_stream ++ spacer

filter(in_pipe)
```

Figure 7.11. Example code. A short example shown as in a user visualization. The filter function is defined, which takes a string parameter and returns the string concatenated with a spacer at either end. The function is then called.

The hyperlinks are replaced with standard identifiers recognizable by the compiler in the following steps:

1. Replace each hyperlink with a newly generated unique identifier name.

2. Generate a type definition for each hyperlinked value.

3. Generate code to associate the type definitions with the identifiers.

4. Concatenate the generated type definitions with the rest of the source code.

The example program has three hyperlinks and in the first part of the transformation each hyperlink is replaced with a new, unique name (step 1).

```
filter = function(in_stream: string) → string
   hyperlink1 ++ in_stream ++ hyperlink1

 filter(hyperlink2)
```

hyperlink1 replaces *spacer* and *hyperlink2* replaces *in_pipe*. Introducing these new identifiers into the code in a strictly typed language requires type coercion (step 2). Most of the code generated by the following transformations is concerned with this task.

The code fragment above, containing undeclared identifiers, is made into legal code by including declarations for the new names, *hyperlink1* and *hyperlink2*. These names do not define new values, instead the declaration must associate the new names with the existing values referenced by the hyperlinks. The hyperlink values are not currently in the scope of the transformed source. They will be introduced as parameters to the *generator* function, which is elaborated on in the next section (Sec. 7.3.1). The result of introducing the values in this way is that they are in scope and typed as *any*, the infinite union type.

In the code, the new identifier names are the actual type of their associated value rather than the infinite union type. Hence, the first step is the generation of code defining the value's type. The second step is the projection of the value onto the new type. The final step is the declaration of the new identifier as the result of the projection.

The first two lines of the generated code in Figure 7.12 are the definitions of the types of the new hyperlink values (red). The type definitions are obtained using reification of the type system. That is, the Hypercode system applies the *typeOf* function (Sec. 3.3.2) to the infinite union typed values. It returns a code string representing the type.

Lines 4 to 9 contain the code to project the hyperlink values onto their specific types. The values have been introduced as a linked list parameter to the *generator* function. *getFromList* is used to extract a hyperlink value with the given name from the list, which is a list of name, value pairs. The *project* statement casts the value from the infinite union type onto the type corresponding to the hyperlink and defined above (lines 1 and 2). That is, the identifier with name *hyperlink1* is assigned the value projected onto type *hyperlink1_type*, which is *string*. The definition of the *getFromList* function is not shown

```
    ! Type declarations
1    type hyperlink1_type is string
2    type hyperlink_type is string

    ! Define the function getFromList (only part shown here)
3    getFromList = function(name: string) → any
      ...

    ! Projections - fetch the values and cast them onto their correct types
4    hyperlink1 = project getFromList("hyperlink1") as X onto
5       hyperlink1_type: X
6       default: nil (hyperlink1_type)

7    hyperlink2 = project getFromList("hyperlink2") as X onto
8       hyperlink2_type: X
9       default: nil (hyperlink2_type)

10   filter = function(in_stream: string) → string
11     hyperlink1 ++ in_stream ++ hyperlink1

12   filter(hyperlink2)
```

Figure 7.12. Example. Include type declarations and projections for the new identifiers (lines 1 to 9).

here. It finds a value in a linked list according to a given key (e.g. `hyperlink1`). The *default* clause (lines 6 and 9) is executed when the hyperlink value is not of the correct type. It returns a nil value of the correct type. This clause would only be executed in the case of an error in the Hypercode system causing it to generate the wrong type for a value, in which case an exception would be raised.

The code in Figure 7.12 forms a correct program where the hyperlinks are replaced by normal identifiers and can therefore be compiled with the standard compiler.

The generator Function

In the evaluation of a Hypercode graph, existing values referenced by hyperlinks are bound into the executable code after compilation by the *generator* function. It encloses the entire program code and takes a list of values as its parameter. This list is created

by the generative engine (coloured blue in Fig. 7.10) and associates values with the identifier names that have replaced the hyperlinks.

Figure 7.13 shows the type of the list, which is a record. It has an *id* field to hold the name and an *entity* field, of the infinite union type *any*, to hold the value. The use of a list is an implementation decision. The *getFromList* function used in Figure 7.12 operates over a list of this type.

```
type list_type is record( id: string, entity: any, next: list_type )
```

Figure 7.13. Example. The type definition of the list of hyperlinks

In Figure 7.14, the *generator* function definition (coloured red) is added around the code. It takes as a parameter the linked list of hyperlink values. The *generator* function is so called because it generates an executable version of the Hypercode (called *wrapper*).

The *generator* function is invoked from the context of the Hypercode system and passed the list of hyperlink values as its parameter in order to bind them into the program. Because the Hypercode system is a fixed context, *generator* must have a fixed type. Its return type is *function()* $\rightarrow$ *any*, a function which returns a value of type *any*. This function is the *wrapper*, defined at the end of the code (red).

wrapper encloses the original Hypercode, so that it can later be executed alone without the overhead of projections during the execution of *generator*. Its return type (*return_type*) is the same type returned by the original code. This type is obtained by compiling the code in Figure 7.12.

After the definition of *wrapper*, a function is defined which calls *wrapper* and returns the result inside an infinite union type. This function is the value returned by *generator*. *generator* should return the *wrapper* function, so the Hypercode can be executed without the projections overhead, but the type of *wrapper* is only discovered during the transformation. Therefore the call to *wrapper* is enclosed in a function which returns a value of type *any*. The final line in Figure 7.14 is *generator*, hence the result of executing the code is the *generator* function itself.

```
type list_type ...
generator ← function (list: list_type) → function() → any {
    getFromList ← function (id: string) → any ... ! get a value from the list

    type hyperlink1_type is string
    type hyperlink2_type is string
    hyperlink1 ← project ...
    hyperlink2 ← project ...

    type return_type is string
    wrapper ← function () → return_type {
        filter ← function (in_stream: string) → string
            hyperlink1 ++ in_stream ++ hyperlink1
        filter ( hyperlink2 )
    }

    ! generator returns a function which calls wrapper
    function() → any
        any ( wrapper() )
}
generator
```

Figure 7.14. Example. The *generator* and *wrapper* functions enclose the program.

Generator Rules

The generative engine is defined by the following set of rules. Red labels refer to the transformed code that results from applying a rule.

1. Replace each hyperlink in the code with a newly generated unique identifier name, where *id* is the hyperlink's ID number. A

   `hyperlink` ⟶ `code ++ hyperlink`*id*` ++ code`

2. Generate a type definition for each hyperlinked value using the *typeOf* function to obtain the type. B

   `hyperlink` ⟶ `type hyperlink`*id*`_type is `*typeOf*`(value)`

3. Generate code to associate the type definitions with the identifiers. *generateNilType* creates an empty value of the correct type for the error case. C

```
hyperlink ⟶
          hyperlinkid = project getFromList ('hyperlinkid') as X onto
              hyperlinkid_type: X
              default = generateNilType(typeOf(value) )
```

4. Create *wrapper* definition. Compile the result of the first three rules to find the return type of the code. D

```
type return_type is typeOf(compile( B + C + A ) )
wrapper ← function () → return_type {
    A
}
```

5. Create *generator* definition.

```
type list_type ...
generator ← function (list: list_type) → function() → any {
    getFromList ← function (id: string) → any

     B
     C
     D

   ! generator returns a function which calls wrapper
     fun() → any
           any ( wrapper() )
}
```

Binding Hyperlinks

Having completed the transformation, the generated code is compiled and executed - this is the binding step in Figure 7.10. Executing the *generator* function causes the hyperlink values to be bound into the program. The Hypercode can then be executed as the *wrapper* function. This section explains the code in the Hypercode system, which acts on the transformed Hypercode to execute it.

When the Hypercode system executes the code in Figure 7.14, it obtains the *generator* function. The execution of *generator* returns a function, and executing this function returns the result of executing *wrapper*, which is a *string*, inside an *any*. Executing the function returned by *generator* is equivalent to executing the original Hypercode program.

```
   !call the compiler with code (the transformed Hypercode)
1  compilation_result ← compile (code)

   !project the result of compilation onto the type of generator
2  project compilation_result.result as X onto
3  function () → any: {
4      Y ← X () ! Y is the generator function

5      project Y as generator onto
6      function (list_type) → function()→ any: {
7          hypercode_function ← generator (list)
8          hypercode_result ← hypercode_function()
9          explode (hypercode_result)
10     }
11     function (list_type) → function (): {
12         hypercode_function ← generator (list)
13         hypercode_function ()
14     }
15     default: raise exception
16 }
17 default: raise exception
```

Figure 7.15. Compilation. Compiling and executing the Hypercode

Line 1 of Figure 7.15 shows the compiler being called with *code*, the transformed Hypercode, as its parameter. Note that this code is part of the Hypercode system as opposed to the previous examples, which were code from the Hypercode graph being evaluated. The compiler returns a structure with the executable function in *result*. In the first *project* statement (line 2) the result of compilation is projected onto type *function()* → *any* and executed. The execution returns *Y*, which is the *generator* function wrapped in an infinite union type. This is in turn projected onto the specific type of *generator*, and then executed. The parameter *list* passed to *generator* is the list of hyperlinks built up during the transformation. Executing *generator* gives a function, *hypercode_function* (line 7) and executing this is equivalent to running the original Hypercode. The return value of the Hypercode is declared as *hypercode_result* (line 8), which has type *any*. To extract a value of the correct type from *hypercode_result*, the Hypercode operation explode (Sec. 7.3.2) is used to reify the value (line 9).

The second part of the project statement (line 11), where Y is projected onto *function (list_type)* $\rightarrow$ *function()*, is used when evaluating a Hypercode program that does not return a value. In this case, *generator* returns a function that returns nothing. The third part of the project statement raises an exception and is chosen when neither of the previous types matches the value. This would only occur if there were an error in one of the generation functions in the Hypercode system, in which case an exception would be raised.

7.3.2 Explode: Generating Hypercode Representations from Values

The explode operation returns a Hypercode graph of the current value of a program element (Sec. 3.6.3). A meta-program explodes a hyperlink via the *getGraphFromHyperlink* operation provided by the MPF.

The Hypercode system is faced with a value from which a Hypercode graph must be generated. This can be done by firstly determining its type and then using a technique for representing values of that type. Code values (including ArchWare ADL functions, abstractions and behaviours) and data values are considered as separate cases. A value's type can be determined using the reflective facilities that support dynamic type discovery. The value, initially represented as an infinite union type, can then be projected onto its specific type.

Code Values

For code values, an HCR is generated at compile time and stored as part of the closure (Fig. 7.6). If a value is determined to be code then:

1. Project the value onto its correct type.
2. If the type is a code type:
   (a) Get the HCR from the closure and
   (b) Return the *source* part of the HCR.

The *source* is a string of XML. An MPF operation parses it to produce a graph data structure that is returned to the meta-program.

Data

If it is determined that the value being exploded is a data value then there are two cases: it may be either a base type or a complex type. An exploded representation of a base type is obtained by displaying the value itself. Base types and an example of their Hypercode representation is shown in Table 7.1.

Type	Representation
`integer`	`1`
`real`	`1.0`
`boolean`	`true`
`string`	`'a string'`

Table 7.1. Exploding Base Types. Base types and examples of their exploded representations.

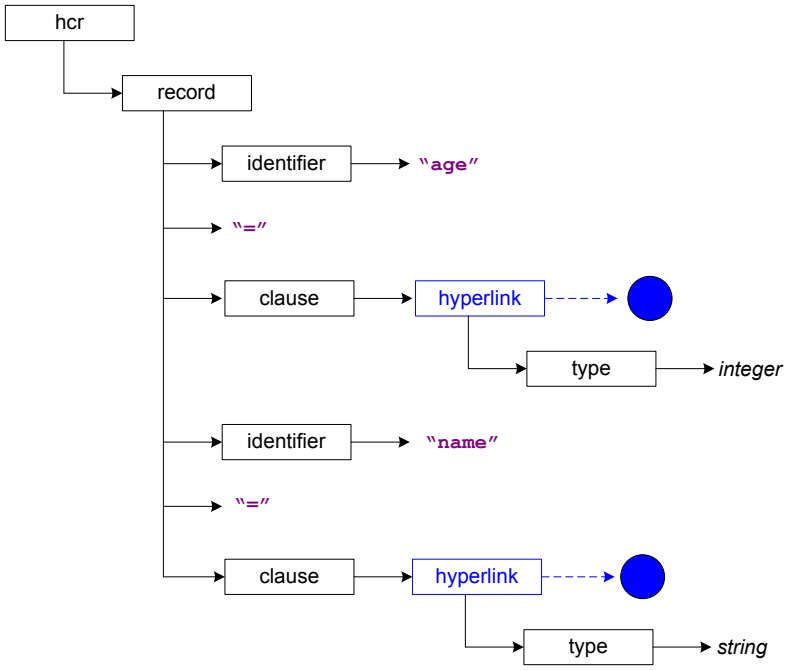
In contrast to base types, that can be represented by a single string, complex types are exploded one level at a time. For example, a record is exploded to show a definition with hyperlinks to each of its elements (Fig. 7.16). Exploding these hyperlinks, which according to their types reference base values, would show that the name is *Bob* and the age is *29*. Other complex types may have more than one level and recursive types may have an infinite number. When exploding cyclic structures, exploding a part of the structure that has already been visited will return the same representation as before, but the Hypercode system uses a stored representation rather than re-generating it.

Because complex types can be user defined, the algorithm for exploding them involves generating code on the fly. This is because code cannot be written to project a value onto its correct type if that type is not known. The type can be obtained dynamically and then the code to project the type can be generated. A similar technique is used in object browsers for PS-Algol (Dearle and Brown, 1988), Napier88 (Kirby and Dearle, 1990) and Java (Kirby and Morrison, 1997).

As an example, the following algorithm is used to generate the representation of a *record* value (r). Initially r is typed as the infinite union type.

1. Get the type representation (t) of r. t is a data structure with a label determining it to be a record and a list containing the types and names (n) of the record's elements.

2. Generate a code string (s) that defines the type represented by t.




```
view ( age = hyperlink, name = hyperlink )
```

Figure 7.16. Exploded View. An exploded view with two parts (name and age) and hyperlinks referencing their values.

3. Generate code projecting r onto t using s. Inside the project statement select an element ($e = X.n$) from the record and return it.

```
project r as X onto
    s:  any (X.n)
    default : Error in generation code - raise exception
```

4. Compile and run the generated code. Assign the result to e.

5. Create a hyperlink referencing e.

6. Repeat for all elements in the record. This gives a list of hyperlinks referencing all the elements.

7. Generate an XML representation of a record including all the hyperlinks as shown at the bottom of Figure 7.16.

The hyperlinks to the individual elements do not need to be created one at a time, as in this example. Instead the code (in step 3) could return an array.

7.4 Operations

MPF operations traverse, manipulate and evolve Hypercode graphs. Their implementation is extendible in that they can be augmented and evolved in the same way as other Hypercode values. The implementation of the operations and the process of evolving existing operations and adding new operations to the framework is explained.

The implementation of the majority of these functions is a straightforward manipulation of the graph data type and will not be described further. Some operations are described here because their implementation is not necessarily obvious.

7.4.1 Traverse

getAllHyperlinks

getAllHyperlinks performs an in-order depth first search and returns a list of all the hyperlink nodes in a graph in program order.

getFirstHyperlink

This operation performs an in-order depth first search of the graph to find the first hyperlink in source code order. The implementation simply uses the *getFirstTagWithName* operation.

getFirstTagWithName

This function returns the first program syntax node in a graph which has the given tag name. This function performs an in-order depth first search of the graph structure to recreate program order.

This search does not follow hyperlinks, they are treated as terminal nodes. An alternative implementation could use the *getGraphFromHyperlink* operation to expand the search space to include hyperlinks.

getGraphFromHyperlink

This operation gets the Hypercode graph of the value referenced by a hyperlink. Its implementation gets the hyperlink value from the HCR data structure and then explodes the value to get the Hypercode graph, as described in Section 7.3.2.

getGraphFromValue

Obtaining a graph from a value is the same as exploding the value. This function is implemented using the explode operation in the Hypercode system (Sec. 7.3.2).

getValueFromHyperlink

Dereference a hyperlink and returns the value. The function finds the enclosing HCR and searches through its list of hyperlinks for the element with the required hyperlink ID.

getXMLFromGraph

getXMLFromGraph returns the XML representation of a graph. In an in-order depth first search, program syntax nodes are converted to XML tags and program text nodes to text strings, i.e. source code.

isEqualNode

The *isEqualNode* returns true if two graphs produce identical programs. This is in contrast to *isSameNode* which returns true if two graphs are the same value. *isEqualNode* uses *getXMLFromGraph* to produce an XML representation of both graphs and then compares the resultant strings.

7.4.2 Manipulate

copyHyperlink

copyHyperlink extracts the hyperlink ID and the HCR ID of a hyperlink node and creates a new graph where the hyperlink is a child of the HCR (with the given IDs). The hyperlink can then be pasted into a new graph using the *pasteHyperlink* operation.

newHyperlinkElement

newHyperlinkElement creates a new hyperlink. Its parameter is a value that will be referenced by the hyperlink. The function creates the new hyperlink and places it in a new HCR data structure. A Hypercode graph of the HCR containing the hyperlink is returned.

pasteHyperlink

pasteHyperlink inserts a hyperlink as a child of the given graph node. It also adds the value referenced by the hyperlink to the list in the graph's HCR. This places the hyperlinked value in the closure of the graph.

updateHyperlinkLocation

Generate a Hypercode graph to update a location referenced by a hyperlink. The new value of the location is represented by a Hypercode graph. The resultant graph must be evaluated using the *evaluateGraph* operation for the update to take place.

7.4.3 Operations for Evolution

evaluateGraph

Evaluating a graph involves compiling and executing the graph. The function implementation uses the evaluate operation in the Hypercode system defined in Section 7.3.1. *evaluate* operates on a string of code containing hyperlinks, so the graph is first converted to XML using *getXMLFromGraph*.

7.4.4 Evolution Patterns

The following operations were developed for the evolution pattern described in Section 5.4 and are specific to the ArchWare ADL.

newDecomposeGraph

A meta-program decomposes a component in order to evolve its components. This function takes as its parameter a component and creates a Hypercode graph to decompose it. The Hypercode graph is a program representation of the decomposition.

The value is not decomposed until this graph is evaluated. The implementation of *newDecomposeGraph* simply creates the graph for a decompose statement and inserts a hyperlink to the component value at the appropriate place.

decomposeGraph

decomposeGraph performs the evaluation of the graph produced by *newDecomposeGraph*. The graph could also be evaluated using the standard *evaluateGraph* operation. However, if the meta-program is going to recompose the same set of components at a later stage then the components' communication channels will need to be reconnected in the same way as before decomposition. In this situation, the *decomposeGraph* operation is used.

decomposeGraph returns the sequence of components resulting from decomposition as well as a list of the connection pairings in the original composition. This list can be used by *newComposeGraph* or *newComposeReplaceGraph* to compose the components and connect their original communication channels.

Connections in the ArchWare ADL are unified within compose statements making this operation necessary. Another language implementing composition and decomposition may take another approach enabling hyperlinks to connections and making the retention of information about communication channels unnecessary.

newComposeReplaceGraph

newComposeReplaceGraph generates a Hypercode graph for a compose statement. It works on the basis that it is constructing a re-composition of a component originally decomposed using the *decomposeGraph* operation. It composes the same set of components as in the original composition, with one being replaced by a new component. The same connections are unified in the composition as were unified in the original component. The parameters to the function are:

- The decompose sequence obtained as a result of the decomposition.
- The saved connections collected by the *decomposeGraph* operation.
- The new component that is replacing one of the components in the original composition.

- The label of the component being replaced.

The following algorithm is used to generate the graph:

1. For each of the components in the decompose sequence loop
2. Get the hyperlink referencing the component value.
3. Get the component's label.
4. If the label is the label of the component being replaced then
5. Create a clause assigning the label to the new component.
6. Else create a clause assigning the label to the hyperlink from 2.
7. End loop.
8. For each pair of connections in the list of saved connections create a clause to unify them.
9. Join the labelling and unification clauses together and insert them into a compose statement.
10. Create an HCR for the graph containing the hyperlinks to the components.
11. Return the graph.

The *isAbstraction* operation is used when including the components to determine if an abstraction must be applied to obtain the component.

This operation has a shortcoming, in that if the component to be replaced has new connection names the connections will not be correctly unified. However, it could be extended to include changed connection names if a meta-program required that functionality. The operation could also be extended to replace more than one component in the composition.

7.4.5 Extending the MPF

All the operations delineated here have been implemented in ArchWare ADL. They are retrieved using a *google* function, which takes the function's name and returns its value (typed as *any*). This value can be exploded to access its Hypercode graph.

Hence, all these operations are available to a meta-program that may use the function value to perform an MPF operation or access the Hypercode graph to evolve the function. A meta-program may also evolve the framework by adding new operations by placing them in the store.

7.5 User Interface

A user interface to Hypercode has been written to provide a visualization of Hypercode for programmers. It also allows the operations *evaluate*, *explode* and *edit* to be applied to Hypercode. Because the Hypercode graph representation is available in XML, it is straightforward to develop interfaces in addition to the meta-programming one. The user interface has most of the functionality available to meta-programs, the difference being that programmers cannot directly access values, or operate over generic program structures in the same way as meta-programs can using evolution patterns.

The user interface has been developed to work together with the Tower Browser (Sec. E.1) to evolve an ArchWare environment. For historical reasons, the user interface in this implementation is not built on top of the MPF as depicted in Figure 7.1. Instead it interacts directly with the Hypercode system.

7.5.1 Edit

An example of some ArchWare ADL Hypercode being edited in the user interface is shown in Figure 7.17. The editor has the usual functionality of a text editor, with additional functionality to accommodate hyperlinks, which are displayed in blue underlined text. A panel at the bottom of the editor is used to display messages about current status or progress to the user.

7.5.2 Explode

Clicking on a hyperlink in the editor *explodes* it. The top of Figure 7.18 shows an editing window in the user interface with a hyperlink. Exploding the hyperlink gives the second window, showing a Hypercode representation of the value referenced by the hyperlink. The example shows the hyperlink exploded to two levels. The first click would have resulted in a *sequence* containing three hyperlinks being displayed.

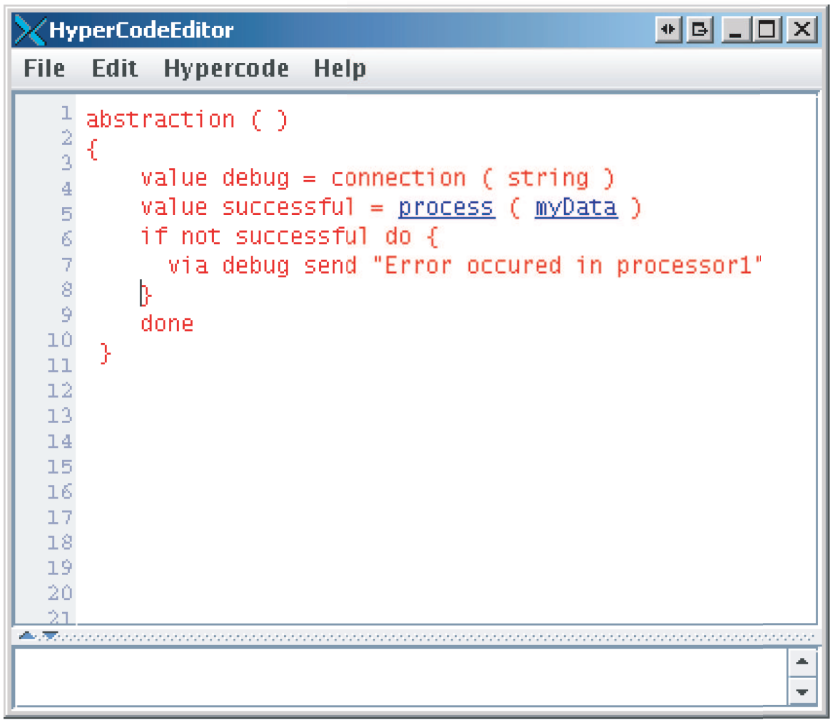


Figure 7.17. Edit. The user interface allows editing of Hypercode programs.

Then clicking on the first of those hyperlinks explodes it as well, giving the Hypercode shown in the picture. Hyperlinks can be imploded by clicking on the red coloured, exploded text.

7.5.3 Evaluate

Hypercode programs can be evaluated by choosing the *evaluate* menu item from the window menu. The result of evaluation is either a hyperlink to the value created, or if the evaluation did not return a value, a message indicating whether the evaluation was successful.

7.6 Summary

This chapter focuses on the implementation of the MPF. It starts with a description of the framework structure (Fig. 7.1). At the top level, a meta-program uses the MPF

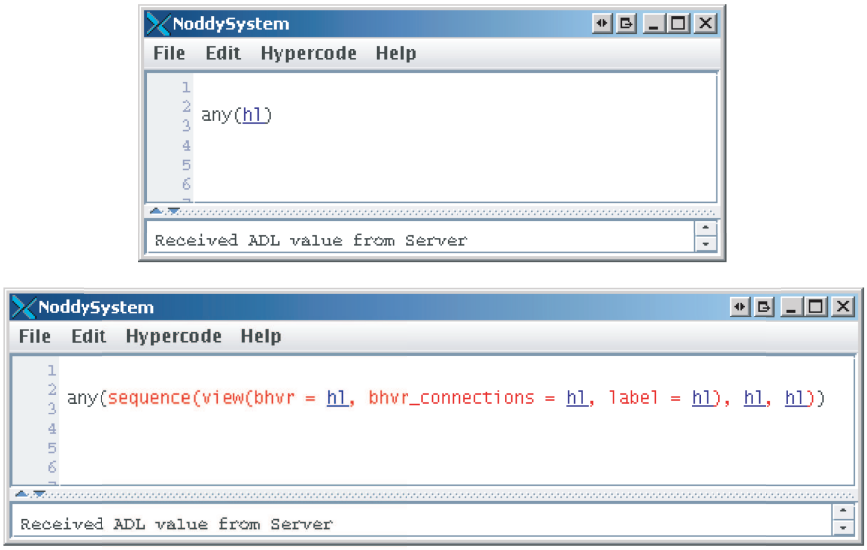


Figure 7.18. Explode. A hyperlink in the first picture is *exploded* to obtain its Hypercode representation in the second picture.

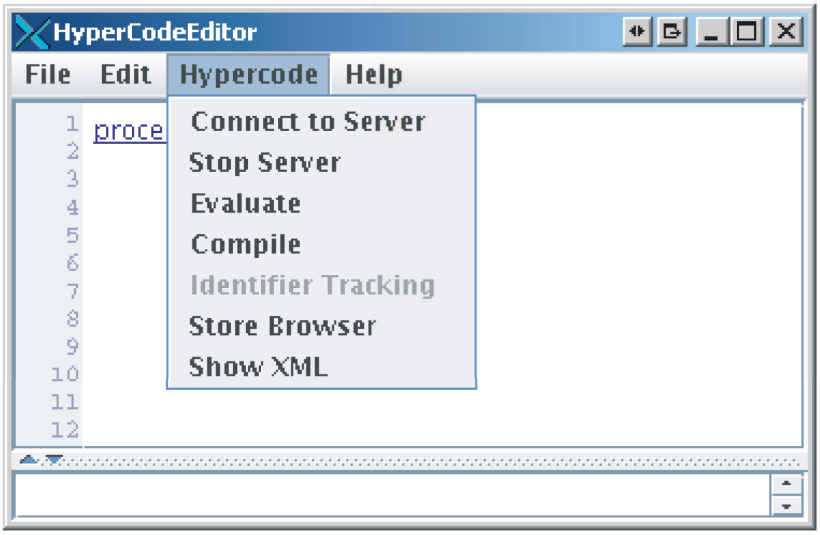


Figure 7.19. Evaluate. A Hypercode program is evaluated using the *evaluate* menu item.

interface consisting of Hypercode graphs and a set of operations that can be applied to them. The interface relies on a Hypercode system to provide the Hypercode operations evaluate and explode. The system is also utilized by a user interface that exposes Hypercode graphs and operations to the developer. Hypercode graphs are stored as Hypercode Representations (HCR) inside the system.

7.6.1 Data Structures

The program representation in the MPF aims to:

- Allow fine grained manipulation by meta-programs.
- Provide an association between values and their representations.
- Be extensible and reusable.

A Hypercode graph is an abstract syntax tree with hyperlinks that reference values in the data graph. In the Hypercode system, Hypercode graphs are represented by HCRs. An HCR is a record, where the graph is stored as XML. Attributes in hyperlink tags reference values in the other part of the record, which is a list of values. The XML representation allows HCRs to be exploited to create the user interface and provide an alternative way of manipulating Hypercode.

HCRs for code values are stored in the closure of the object that represents them. The HCRs of data values are generated on the fly using the explode operation in the Hypercode system. This ensures that variable values always have an up to date representation.

7.6.2 Hypercode System

Generative programming is used to implement the Hypercode operations: evaluate and explode. The evaluate operation transforms, compiles, binds and executes some Hypercode. In order to use a standard compiler as part of this process, the Hypercode is transformed into an equivalent program without hyperlinks. The values referenced by the hyperlinks must then be bound into the compiled program. The evaluate process is as follows:

1. Transform a Hypercode graph into XML code.
2. Wrap the code in a generator function that takes a list of the values referenced by hyperlinks as its parameter.
3. Compile the generator code
4. Bind the hyperlinks into the compiled program by executing the generator function and passing it a list of hyperlinks.
5. Executing the resultant value is equivalent to executing the original Hypercode.

Explode generates a Hypercode graph from a value. The HCRs of code values are stored in their closure. Therefore, they can be exploded by accessing the HCR and parsing the XML to generate a Hypercode graph structure. Data values are exploded by:

- Reifying the value's type using the *typeOf* function.
- Projecting the value onto this type.
- Then generating code to represent the value as that type.

User defined types can be broken down into a series of types and exploded one level at a time.

7.6.3 MPF Operations

The operations in the MPF can be applied to traverse, manipulate and evolve Hypercode graphs. An evolution pattern has been developed that relies on support for composition and decomposition in the ArchWare ADL. The pattern allows a meta-program to evolve a Hypercode graph with only limited knowledge of its internal structure.

7.6.4 User Interface

The user interface supports the Hypercode operations evaluate, explode, implode and edit and allows developers to perform the same evolutionary processes on Hypercode as the meta-programs can using the MPF. The data structure used to represent Hypercode for the MPF is re-used in the user interface.

Chapter 8

Conclusion

This chapter concludes and summarizes the thesis and discusses future work.

The unique combination of technologies in the Meta-Programming Framework may be used for the automatic evolution of long-lived, complex systems with emergent properties. A decomposition mechanism for partially stopping the system means that components being evolved can be isolated from the rest of the executing system. A representation of program closure allows meta-programs to make changes to a complete and current model of the dynamic state of the components. Evolved components can then be reintroduced to the executing system using structural reflection.

The main contribution of this thesis is a set of mechanisms for automating evolution. It presents a meta-programming interface to executing systems and a process for incrementally evolving them that involves a novel combination of technologies: decomposition; a representation for executing programs; and structural reflection.

These technologies support the process for incrementally evolving systems depicted in Figure 8.1. A meta-program stops part of the system by decomposing it (1). It retrieves a representation of the parts of the decomposed component (2) in the form of a Hypercode graph. The Hypercode graph is updated, or replaced, by the meta-program and a new component value is created using structural reflection (3). In this example, one of the circle type parts of the component has been replaced by a star type. Note that the hyperlinks, shown as blue arrows, refer to the same values in the executing system before and after evolution. In this way, internal state can be preserved over the changes. The final step is placing the new value into the executing system (4).

8.1 Summary

The work in this thesis takes place in the context of evolving software systems, which continue to execute as some part of the system is stopped, changed and restarted by a management component referred to as a meta-program. An analogy is made between control systems and evolving software, where feedback controllers correspond to closed adaptive engines and feedforward controllers to open adaptive engines. A closed adaptive engine uses internal information to adapt existing functionality and is often implemented using probes and gauges. An open adaptive engine introduces novel functionality into the system. The MPF supports the introduction of new functionality by an open adaptive engine, but does not answer the question of how to gather and interpret environmental input.

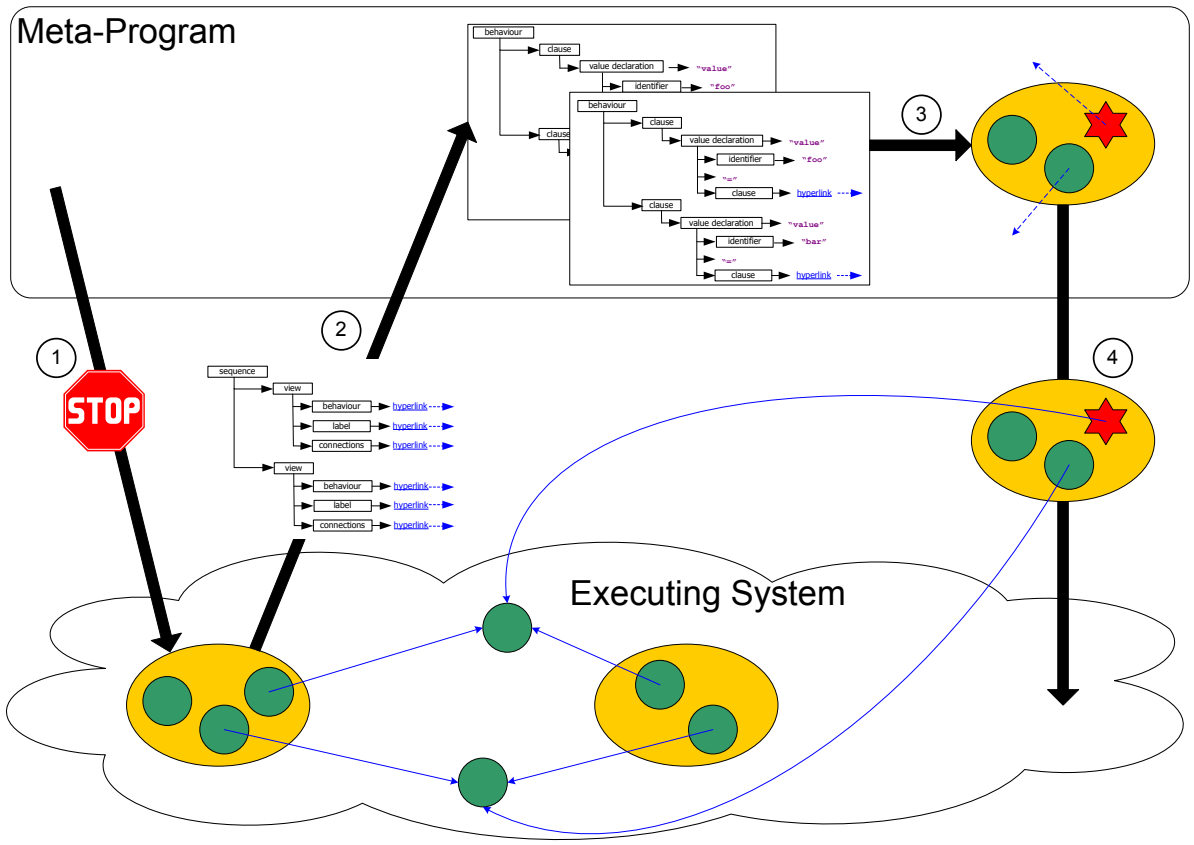


Figure 8.1. Evolution Process. A meta-program stops part of an executing system (1). It gets a representation of the system (2) and makes some changes to the representation before evaluating it to get a new value (3). The new value is placed in the executing system (4). Note that internal state and data is maintained over the changes (blue arrows) and other components continue to execute uninterrupted.

Software architectures approach system design from a high level of abstraction, modelling software in terms of its components and their interactions. In an evolving system, a software architecture assists understanding about the effects of evolution on the system as a whole. Constraints on software architectures are often used to direct evolution and ensure that it does not violate original design intentions.

There is a spectrum of techniques for evolving software, from fully automatic to user driven. Simple automatic changes, e.g., updating a variable, are used in most systems. More complex changes, such as altering existing components or adding new

components, are difficult to understand and apply, necessitating developer intervention. Meta-programming is a way of making complex changes automatically. In a software architecture, meta-programs act as change management components that direct and apply evolution in other components.

Evolving software systems can be categorized in terms of the aspects of evolution they support. Characteristics to consider include:

- *When* evolution takes place.
- *What* changes can be made.
- *How* the changes are applied.
- *Who* drives the evolution process.
- *Where* changes take place.
- Whether there is a structure for *change management*.

In an ideal framework change is effected with minimal disruption, evolutions are applied to an accurate model of the system, unrestricted changes may be applied, and new components can incorporate the system's extant state.

The MPF defines Hypercode graphs, a representation that captures the program closure of an executing system. Hypercode graphs give meta-programs an accurate model of the system to which evolutions can be applied. The MPF operations support unrestricted changes to Hypercode graphs, which can be integrated back into the system using structural reflection. New and updated components can include extant values and therefore preserve system state over evolution, because their Hypercode graphs are a complete model including program state and data. In the MPF, Hypercode graphs and structural reflection are used in combination with decomposition, which stops a small part of the system so it can be evolved. Consequently, the MPF provides the characteristics of an ideal framework for evolution.

Other frameworks for software evolution concentrate on different aspects of the problem. ADLs have been developed to model evolving software architectures. Some examples of these are Darwin, Gerel and Weaves. ArchJava integrates a software architecture with program source code. This is one way of making sure the implementation is an accurate reflection of the software architecture as it evolves. The same goal is

achieved by frameworks that include tools to mediate between a software architecture model and the executing system. These tools apply changes to both so that they stay in sync. Examples are ArchStudio (Sec. 2.6) and Rainbow (Sec. 2.5). A system that concentrates on mechanisms for applying change, similarly to the MPF, is Intentional Programming. It uses a program representation incorporating source code and data. Meta-programs perform transformations on this representation to, for example, compile or display it.

In the MPF, structural reflection provides the ability to introduce new values into the current environment. Hypercode facilitates introspection and an abstraction over the difference between code and data. With introspection, a meta-program can obtain a Hypercode graph representation of any value in the executing system in order to examine its current state and / or evolve it. Abstracting over the difference between code and data implies that the Hypercode graph representation of a code value takes the same form as that of a data value. ProcessBase, the platform used to implement the MPF provides first class functions as well as orthogonal persistence and structural reflection. Support for modelling and evolving software architectures in the MPF is provided by ArchWare ADL. In particular the decompose operation, that stops part of an executing system, is necessary for the incremental evolution process.

A Hypercode graph uses an amalgamation of an abstract syntax tree and a data graph to represent program syntax and program closure. A set of operations on Hypercode graphs defines an interface for meta-programs. The operations allow meta-programs to create and change Hypercode graphs, which is equivalent to writing and editing programs. Operations to support an evolution pattern where a component in a composition is updated have also been produced.

Using the evolution pattern defined in this thesis, a meta-program can update a component in a composition following the process that was illustrated in Figure 8.1. An operation is available to decompose a composed component and retain information about its internal connections. To recompose after changes have been made, another operation uses the information about connections to reconnect the components as they were before the evolution. The internal state of components in the composition is retained over the evolution by virtue of hyperlinks in the Hypercode graphs.

The implementation of the MPF is both extensible and reusable. The MPF operations are extensible because they are programmed in the MPF and can therefore be evolved in the same way as other values. New operations can be created and the existing ones

changed. Hypercode graphs are implemented using a general framework that could be applied to other platforms. This is demonstrated by the adaptation of a ProcessBase Hypercode framework to an ArchWare ADL framework. The Hypercode graph representation can also be re-used as demonstrated by the user interface, which allows a programmer to interact directly with Hypercode.

The parts of the MPF implementation are:

- A data structure to represent Hypercode.
- MPF operations that allow meta-programs to interact with Hypercode graphs.
- The Hypercode operations evaluate and explode, implemented using generative programming.
- A user interface for evolving Hypercode.

8.2 Discussion

The MPF provides an API that allows meta-programs to create, introspect and evolve components. It is combined with other technologies to provide the mechanisms for incremental evolution. Building software that can evolve incrementally allows the software to be adapted concurrently with the business processes that use it.

8.2.1 Evolution Patterns

The changes a meta-program can make to another component using the MPF are only restricted by the capability of the programming language and semantic correctness. Therefore, the MPF can be used as part of an open adaptive engine, which introduces new functionality into the system and reacts to emergent behaviour. The advantage of this approach is that the MPF model is suitable for use by a wide range of applications. Applications can decide their own change management procedures and apply these on top of the MPF. The disadvantage is that, unless restrictions are applied, evolutions are complex to define and difficult to automate.

Future work on the MPF includes the definition of evolution patterns to be used as part of a change management framework. Change management components can then

choose appropriate evolution patterns to apply under certain conditions. This thesis defines one pattern, the update of a component in a composition, and the MPF operations to support it. Defining more patterns would involve determining common evolutions, defining the processes by which they take place, and writing the MPF operations to support those processes.

In addition to using evolution patterns, a change management system has to control conflicts that may be caused in an evolving system. For example, parallel evolutions may result in an inconsistent system. Change management is responsible for the synchronization of evolving components that use shared data.

8.2.2 Framework Evolution

The MPF has been constructed in itself and is able to be evolved in the same way as any other value. Therefore, new MPF operations can be created, for example, in order to define a new evolution pattern. Existing operations can be specialized or updated to reflect the needs of a particular application. Hypercode graphs can be extended to, for example, include software architecture information and constraints.

There are a number of difficulties in evolving the framework. Writing new operations is not trivial because it requires an understanding of the underlying data structure, namely, the Hypercode graph. Making changes to existing operations has an effect on programs that depend on those operations, raising issues of system safety and correctness.

MPF operations can be evolved dynamically. In the implementation, programs fetch an MPF operation from a mutable location before using it. Therefore, replacing an operation simply involves updating the value in the mutable location. The limit on evolution of the standard MPF operations is the evaluate and explode mechanisms provided by the Hypercode system. These are not developed in the MPF and cannot be represented as Hypercode graphs. The system must be stopped in order to change them. This is because evaluate and explode use a number of tools developed outside the MPF. Firstly, they generate code and compile it. This is clearly necessary for evaluation, and is required in explode in order to introspect user defined types. Secondly, both of them introspect the running system. The MPF explode operation makes it appear as if introspection returns Hypercode graphs of the system. In order to do this, it translates ProcessBase types into ADL and then into Hypercode graphs.

Another value that cannot be evolved through the MPF interface is one of the operations in the evolution pattern presented in this thesis - *decomposeGraph*. It re-unifies connections as they were before a decomposition and relies on internal knowledge of the compiler to find out information about the status of connections at decomposition time.

The information in Hypercode graphs can be evolved by extending or altering the program syntax. Applying the Hypercode graph representation to a new programming language requires no changes to the graph structure. The only difference in the graphs of different programming languages is in the names of program syntax nodes. The evaluate and explode operations that generate Hypercode graphs have to be altered to generate nodes for the new language and to use a new compiler. In order to make these changes, and any that involve altering the Hypercode system, it is necessary to stop the system.

The structure of Hypercode graphs can be extended by adding new node types and using node attributes, which are currently only used for hyperlink IDs. For example, a Hypercode graph can be augmented with software architecture information. Such changes only involve updating the compiler, not the Hypercode system, and can be done on the fly. The compiler is changed to either ignore the extra information, or check it. Then the mutable location containing the compiler can be updated without bringing the system down.

There are two restrictions on the evolution of the Hypercode graph structure. Firstly, the graph should be able to be converted to a string of XML. Secondly, hyperlink tags must retain their ID attribute in order that the Hypercode system can retrieve the values they reference. Additionally, changing the structure of the Hypercode graph may involve altering some or all of the MPF operations. Extending a Hypercode graph by adding program syntax nodes containing new information does not change the structure of the graph and does not affect any of the standard MPF operations.

8.2.3 Degree of Automation

The addition of new MPF operations is equivalent to the introduction of new values into the application and brings up the same difficulties with respect to automation. There is not yet a process by which new components can be defined automatically.

This thesis gave an example of a new function being introduced into the system automatically, but the function, i.e., the code containing new functionality, was defined by a programmer and the meta-program simply read in the file.

Automating change to existing components is a more tractable problem. For example, the update of a component can be automated and if the component conforms to some defined structure, the same generic pattern can be used to update multiple components. An adaptive engine can use probes and gauges to automatically determine which adjustments to apply to existing components by choosing them from a set of possible changes.

8.2.4 Practicality

The manipulation of existing programs is difficult due to the complexity of understanding extant code and how it can be programmed over. Previous experience with string manipulation for structural reflection has indicated that although it is a powerful technique, it is hard to use. Hypercode graphs allow the developer to operate at a higher level of abstraction. This makes reflective programming slightly easier, but is still not straightforward.

The example in Chapter 6 gives an indication of the difficulty involved in developing a meta-program. An understanding of the current structure of the Hypercode graph, over which the meta-program is operating, is required in order to write each line of code. A similar problem is faced by programmers using the DOM interface to manipulate XML documents. This has led to the development of languages such as XPath (W3 Consortium, 1999) and XQuery (W3 Consortium, 2005b). Generic evolution patterns are required to make the MPF more usable.

Consider the existing evolution pattern that replaces a single behaviour in a composed component. It is not hard to understand or apply. The whole example in Chapter 6 could be generalized into an evolution pattern. The pattern replaces a hyperlink, with a given name, in one of the sub-components of a composition, with a new hyperlink. These evolution patterns abstract over the code manipulation and can be usefully applied to an evolving system.

It may not be practical to use the MPF only for multiple, distinct evolutions to a system because of the effort necessary to program each evolution. However, there is a trade off between genericity and complexity. A generic meta-program that is difficult to write

may be worth the effort if it can become an evolution pattern that is used frequently. A change management framework should use a set of evolution patterns to make the most common changes. The MPF has the power to represent generic evolution patterns and the flexibility to allow new patterns to be developed over time.

8.3 Future Work

8.3.1 Technology Transfer

The technologies on which the MPF is based are not widely used. Future work involves transferring the MPF to a well used platform where it could be further evaluated. The Hypercode graphs and operations are based on XML and would not be difficult to transfer. The combination of technologies that support incremental evolution, i.e., decomposition, introspection and structural reflection, can be provided in another language, for example, Java.

An existing implementation of Hypercode for Java (Zirintsis et al., 2001) shows that it is possible to transfer the Hypercode system to Java. This includes introspection and structural reflection. The remaining technology, decomposition, can be implemented using Java. One approach is for components to implement the *Runnable* interface so they can be started, stopped and made to wait. A *Connector* class would be defined to allow components to send synchronized messages to each other. It would use semaphores to make components wait on messages if they have been decomposed. Composition would start a set of components. Decomposition would disconnect the components, using *Connector*, and return a set of handles to them. This approach implements decomposition in the same way as in the ArchWare ADL. Components continue to execute until they reach a reduction limit, where they are either finished, or waiting to send or receive a message. An alternative is for decomposition to stop the execution of the components.

8.3.2 Software Architecture Extraction

A desirable extension to the MPF is an interface to extract the software architecture from the implementation. Access to a model of the software architecture is essential for change management tools that rely on architectural constraints to drive evolution.

The architecture could be extracted by introspection on the executing system. A Hypercode graph representation will contain architectural information if that information is part of the programming language it represents. It could include information about elements in the software architecture as well as constraints and links to analysis functions.

The ArchWare ADL contains explicit architectural constructs, but connections are not first class values. Therefore, architectural topology cannot be determined by inspecting a Hypercode graph based on ArchWare ADL. This may change on application of the technology to a new platform.

8.3.3 Performance

The MPF implementation is not particularly efficient for two reasons. Firstly, it uses generative programming techniques and structural reflection extensively. Secondly, it uses XML, requiring that strings to be frequently parsed to create data structures. There is a trade off between an implementation that is more efficient and one which is transferable and reusable. In applying the MPF to other platforms, it may be appropriate to make the implementation more efficient and therefore more attractive for use as part of an evolution framework.

8.3.4 Integration with Intentional Programming

An interesting MPF extension would be its integration with the Intentional Programming Framework. IP represents a program at development time and would be complemented by the MPF representation that extends for the lifetime of the program. The IP development environment could then be used to develop and evolve programs. The integration would not be straightforward because the existing MPF infrastructure does not map to the IP model.

8.3.5 Multiple Programming Languages

Another possible extension to the MPF, suggested by integration with IP, is using a Hypercode graph to represent a program written in multiple languages. In the same way that generative programming techniques are used to remove hyperlinks, the Hypercode graph could be reduced to a compilable program in a single language. For

example, the inclusion of software architecture information in the source code could be achieved by combining an ADL and another programming language.

8.3.6 Change Management Framework

The MPF needs to be integrated into a change management framework and used to evolve real software systems before claims about its usefulness can be proven. Studying its use would provide insight into how the MPF could be improved. There are undoubtedly many more MPF operations that would benefit evolving systems and new evolution patterns would hopefully emerge over time.

Research on a real application is also necessary to determine how much of the response to emergent behaviour in complex systems can be automated. For example, could the system detect the need for programmer intervention or suggest potential changes?

8.3.7 Conclusion

This thesis has proposed that the Meta-Programming Framework enables the development of open adaptive engines for automatically evolving software systems according to the Principles of Minimal Disruption and Unrestricted Evolution. By definition, neither the behaviour nor the evolution of long-lived systems, with emergent properties, can be predicted at development time. To address the problem, the thesis presented mechanisms that can operate during execution to provide insight into the state of the system and introduce unplanned and unrestricted changes.

Some evolutions, or parts of evolutions, can be completely automated, but programmer intervention will always be necessary to deal with emergent properties and introduce new functionality in open systems. The support of automation provided by the MPF and the change management framework is required to make the programmer's task tractable.

This thesis introduced an innovative mix of technologies to support evolution by meta-programs. It contributes an interface to executing systems and a methodology for evolving them. The technology has been implemented as the Meta-Programming Framework.

Meta-programs are presented with an interface that provides a representation of every value in the system, including itself, in the form of a Hypercode graph. The interface

can be evolved along with the system. Hypercode graphs model executing programs and support continuity by allowing component updates that preserve internal state.

New and existing technologies are combined in an incremental evolution process. Firstly, decomposition stops part of the system. Then using the MPF interface, a meta-program evolves Hypercode graph representations of the relevant components. Finally, the changes are incorporated into the executing system using structural reflection. An experimental example in the thesis demonstrated the incremental evolution process and the Hypercode graph representation by updating an executing system.

A top-down approach to evolving software restricts system design by prescribing particular software architectures for change. The definition of a set of possible changes at development time limits evolution to changes that can be foreseen before it is even used. In contrast, the MPF provides powerful tools for introspection and evolution that concentrate on underlying support for unrestricted change. Change management frameworks can therefore layer their own evolution processes and policies on top and adapt them in the face of emergent behaviour. For example, a change management framework contains the policies of when and how to make change, and ensures the safety of changes. Without this structure, the change enabled by the MPF could be ad-hoc and unsafe. The technology in this thesis provides a useful contribution to the development of frameworks that can automate evolution. It provides an interface for adaptation infrastructures to extract information from and effect changes on systems.

In the future, a set of experiments is required that incorporates the MPF in a change management framework and applies it to a system with real users and changing requirements to reveal its strengths and weaknesses. Claims to the usefulness of the change mechanisms cannot be proven until they have been tested in more than a small example. The process of evolving a real system provides insight into interesting problems of software evolution, such as what changes to make and when to make them.

Appendix A

Meta-Programming Framework CFS

This appendix defines the context free syntax (CFS) of the ArchWare ADL with XML mark-up.

A.1 Context Free Syntax of ArchWare ADL with XML

Hypercode graphs are marked up with syntax information and stored as XML in HCRs. This chapter defines the context free syntax of the ArchWare ADL marked up with XML tags. The standard syntax is extended to include hyperlinks.

Declaration

program ::= <hypercode> description </hypercode>

description ::= declaration [; description] | clause [; description]

declaration ::= <declaration> type_declaration | value_declaration </declaration>

Type Declaration

type_declaration ::= <typedeclaration> **type** type_definition </typedeclaration> |
<typedeclaration> <recursive/> **recursive type** type_definition [
& type_definition]* </typedeclaration>

type_definition ::= <typedefinition> identifier **is** type </typedefinition>

Type Descriptor

type ::= <type> type_def </type>

type_def ::= <integer> **integer** </integer> |
<real> **real** </real> |
<boolean> **boolean** </boolean> |
<string> **string** </string> |
<any> **any** </any> |
<connection> **connection** [[type_list]] </connection> |
<behaviour> **behaviour** </behaviour> |
<abstraction> **abstraction** [[type_list]] </abstraction> |
<identifier> identifier </identifier> |
<view> **view** [identifier_type_list] </view> |
<function> **function** [[type_list]] → <result> type </result>
</function> |

type_def (cont.) ::= <location> **location** [type] </location> |
<sequence> **sequence** [type] </sequence>

type_list ::= type [, type_list]

identifier_type_list ::= identifier : type [, identifier_type_list]

Value Declaration

value_declaration ::= <valuedeclaration> **value** identifier_clause_list
</valuedeclaration> | <valuedeclaration> <recursive/> **recursive**
value identifier_literal_list </valuedeclaration>

identifier_clause_list ::= identifier = clause [, identifier_clause_list]

identifier_literal_list ::= identifier = literal [& identifier_literal_list]

Clause

clause ::= <clause> clause_def <clause>

clause_def ::= <if> **if** clause <then>**then** clause </then> <else> **else** clause
</else> </if> |
<if> **if** clause <do> **do** clause </do> </if> |
<while> **while** clause <do> **do** clause </do> </while> |
<replicate> **replicate** clause </replicate> |
<compose> **compose** {parallel_list [<where> **where** {unification}
</where>] [<free> **free** labelled_identifier_list </free>]
[<precedence> **precedence** precedence_list </precedence>]}
</compose> |
<decompose> **decompose** clause </decompose> |
prefix |
<choose> **choose** { choice_list } </choose> |
<project> **project** clause <as>**as** identifier </as> **onto**
project_list <default> **default** : clause </default> </project> |
expression <assignment> **:=** </assignment> clause |
<iterate> **iterate** clause [<by> **by** identifier : type <by>]
<from> **from** identifier = clause </from> <accumulate>
accumulate clause </accumulate> [<as> **as** identifier </as>]
</iterate> | expression

parallel_list ::= <composedbehaviour> label **as** clause </composedbehaviour>
[<and> **and** </and> parallel_list]

unification ::= <unification> labelled_identifier **unifies** dynamic_identifier
</unification> [**,** unification]

labelled_identifier ::= label **::** identifier

labelled_identifier_list ::= labelled_identifier [**,** labelled_identifier]

dynamic_identifier ::= labelled_identifier

precedence_list ::= identifier > identifier [**,** precedence_list]

choice_list ::= <choice> clause </choice> [<or> **or** **</or>** **choice_list**]*

project_list ::= <onto> type : clause </onto>; [project_list]

prefix ::= <prefix> **via** identifier <send> **send** [clause_list] </send>
</prefix> | <prefix> **via** identifier <receive> **receive**
[identifier2_type_list] </receive> </prefix> | <prefix>
unobservable </prefix>

identifier_type_list ::= identifier_type [**,** identifier_type_list]

identifier2_type_list ::= identifier_type [: type] [**,** identifier2_type_list]

identifier_type ::= identifier : type

Expression

expression ::= **(**clause**)** |
{<description> description </description>} |
<behaviour> **behaviour** clause </behaviour>|
literal |
<not> **not** </not> expression |
expression <and> **and** </and> expression |
expression <or> **or** </or> expression |
add_operator expression |

expression (cont.) ::= expression relational_operator expression |
expression add_operator expression |
expression multiply_operator expression |
expression <concatenate> **++** </concatenate> expression |
expression **(**clause | clause**)** |
expression **(**[clause_list]**)** |
<view> **view (**identifier_clause_list**)** </view> |
clause **.** identifier |
<location> **location (**clause**)** </location> |
'clause |
<sequence> **sequence (**clause_list**)** </sequence> |
<sequence> **sequence** <for> **for** identifier **=** clause </for> <to>
to clause </to> <using> **using** clause </using> </sequence> |
expression **::** clause |
expression <including> **including** expression </including> |
expression <excluding> **excluding** expression </excluding> |
<size> **size (** clause **)** </size> |
identifier |
hyperlink

clause_list ::= clause [**,** clause_list]

identifier_list ::= identifier [**,** identifier_list]

relational_operator ::= equality_operator | ordering_operator

equality_operator ::= <equal> **=** </equal> | <notequal> = </notequal>

ordering_operator ::= <lessthan> < </lessthan> | <lessthanorequal> **<=**
</lessthanorequal> | <greaterthan> > </greaterthan> |
<greaterthanorequal> **=>** </greaterthanorequal>

hyperlink ::= <hl id="label"> <type> <name> label </name>
type_declaration </type> label </hl>

add_operator ::= <plus> **+** </plus> | <minus> **-** </minus>

multiply_operator ::= integer_multiply_operator | real_multiply_operator

integer_multiply_operator ::= <multiply> * </multiply> | **div** | <rem> **rem** </rem>

real_multiply_operator ::= <multiply> * </multiply> | <divide> /</divide>

Literal

literal ::= integer_literal | real_literal | boolean_literal | string_literal | connection_literal | behaviour_literal | abstraction_literal | view_literal | sequence_literal | function_literal

integer_literal ::= <integer> [add_operator] digit [digit]* </integer>

real_literal ::= <real> integer_literal.[digit]*[e integer_literal] </real>

boolean_literal ::= <boolean> **true** </boolean> | <boolean> **false** </boolean>

string_literal ::= <string> "character" </string>

connection_literal ::= <connection> **connection (** type_list **)** </connection>

behaviour_literal ::= <behaviour> **done** </behaviour>

abstraction_literal ::= <abstraction> **abstraction (** identifier_type_list **);** clause </abstraction>

view_literal ::= <nilview> **nilview (** type **)** <nilview>

sequence_literal ::= <nilsequence> **nilsequence (** type **)** </nilsequence>

function_literal ::= <function> **function (** [identifier_type_list] **)** [→ <result> type </result>] **;** clause </function>

Identifier

identifier ::= <identifier> letter[letter|digit|_]* </identifier>

label ::= <label> identifier </label>

Appendix B

Definition of MPF Operations in ArchWare ADL

This appendix defines the MPF operations on Hypercode graphs.

B.1 Graph Operations

The ArchWare ADL MPF operations for Hypercode graphs are defined in this appendix. They support the traversal, manipulation and evolution of Hypercode graph structures.

B.1.1 Hypercode Graph Type

```
recursive type graph is view [node: location[tagType],
                              parent: location[graph],
                              children: location[graphList]]

& tagType is view [name, text: string, attributes: location[attributeList]]

& graphList is view [node: graph, next: location[graphList]]

& attributeList is view [name, val: string, next: location[attributeList]]
```

B.1.2 MPF Operations

addAttribute

Adds an attribute to a node's list of attributes. Only works for element type nodes - not text nodes. The first parameter is the attribute name. The second parameter is the attribute value. The third parameter is the graph element to which the attribute will be added.

Parameter: **string**
Parameter: **string**
Parameter: **graph**

appendChild

Adds a child node to the end of the list of children of this graph node. The first parameter is a graph representing the child node. The second parameter is the graph to which the child will be added. Returns a graph with the child node appended, or the child node itself if the graph was empty.

Parameter: **graph**
Parameter **graph**
Return Type: **graph**

appendChildAt

Adds the node 'child' at the given position in a graph's list of children, starting at position 0. The first parameter is a graph to be added as a child node. The second parameter is the graph to which the child will be added. The third parameter is the position to add the child. Returns a graph with the child node in its list of children, or the child node itself if the graph was empty.

Parameter: **graph**
Parameter: **graph**
Parameter: **int**
Return Type: **graph**

appendChildDescend

Adds the child node to the end of the list of children of the graph node and returns the child. The first parameter is a graph to be added as a child node. The second parameter is the graph to which the child will be added. Returns the child node.

Parameter: **graph** A graph to be added as a child node. Parameter: **graph** The main graph.
Return Type: **graph**

copy

Create a copy of a graph.

Parameter: **graph** A graph to be copied.
Return Type: **graph**

copyHyperlink

Create a copy of a hyperlink node that can be inserted into a different graph.

Parameter: **graph** A hyperlink node to be copied.
Return Type: **graph**

evaluateGraph

Evaluate a graph Returns the result of the evaluation represented as a graph.

Parameter: **graph**
Return Type: **graph**

extractADL

Extract an ArchWare ADL string from an XML representation. The parameter is a string of XML. Returns a string of ArchWare ADL.

Parameter: **string**
Return Type: **string**

getAllNodesWithName

Get all the nodes with the given tag name (type string) in the given graph (in-order depth first search). Returns a list of graphs containing all the names with the given tag in the graph.

Parameter: **string**
Parameter: **graph**
Return Type: **graphList**

getAllHyperlinks

Get all the hyperlinks in a graph (in-order depth first search). Returns a list of graphs containing all the hyperlinks in the graph.

Parameter: **graph**
Return Type: **graphList**

getAttributes

Get the attributeList from a graph node. Returns a list of attributes.

Parameter: **graph**
Return Type: **attributeList**

getAttributeValue

Get the value of this attribute, not applicable to text nodes. The first parameter is the attribute name. The second parameter is the graph element whose attributes are being searched. Returns the attribute value if found, an empty string otherwise.

Parameter: **string**
Parameter: **graph**
Return Type: **string**

getChildNodes

Get a node's list of child nodes. Returns the child node as a `graphList`.

Parameter: **graph**
Return Type: **graphList**

getChildPosition

Get the position of this graph node in the list of children of its parent. Returns an integer representing the position (starting from 0), or -1 if not found.

Parameter: **graph**
Return Type: **int**

getFirstChild

Get the first child of this node. Returns the child graph node.

Parameter: **graph**
Return Type: **graph**

getFirstChildWithName

Get the first child of this node which has the given name. The first parameter is the name of the child node to be found. The second parameter is the graph whose children will be searched. Returns the child node, if found, or an empty graph.

Parameter: **string**
Parameter: **graph**
Return Type: **graph**

getFirstHyperlink

Get the first hyperlink which is a descendant of a graph node (depth first search) - i.e. finds the first hyperlink which would appear in the expanded XML representation. Returns a graph representing the first hyperlink which is a descendant of this graph node.

Parameter: **graph**
Return Type: **graph**

getFirstTagWithName

Get the first graph with the given name which is a descendant of the given graph (in-order depth first search). Equivalent to finding the first tag with this name which would appear in the expanded XML representation. The first parameter is the tag name of the graph element we are looking for. The second parameter is the graph being searched. Returns a graph and a nil graph if not found.

Parameter: **string**
Parameter: **graph**
Return Type: **graph**

getGraphFromHyperlink

Get the graph representation of a value referenced by a hyperlink. Returns the graph or nil(graph) if the value isn't found.

Parameter: **graph**
Return Type: **graph**

getGraphFromValue

Get the graph representing a value. Returns the graph.

Parameter: **any**
Return Type: **graph**

getHyperlink

Get the hyperlink referenced by this node - which has the following type definition:

```
recursive type Hyperlink is view [id, source: string;
                entity: any;
                exploded: location[string];
                isType: boolean;
                references: location[*Hyperlink]]
recursive type HyperlinkTable is view [ hyperlink : Hyperlink;
                                        next : location[ HyperlinkTable ]]
```

Returns the hyperlink or nilview(Hyperlink).

Parameter: **graph**
Return Type: **Hyperlink**

getLastChild

Get the last child of this node. Returns the child graph node.

Parameter: **graph**
Return Type: **graph**

getLocalName

Returns the name of this node. Only valid for program syntax nodes - program text nodes do not have names.

Parameter: **graph**
Return Type: **string**

getNextSibling

Get the node immediately following this node in the list of its parent's children. Returns the child's successor in the list of children or an empty graph if not found.

Parameter: **graph**
Return Type: **graph**

getNodeName

The name of this node, depending on its type. The name, if it's an element node, or the text, if it's a text node.

Parameter: **graph**
Return Type: **string**

getNodeType

The type of this node, which is a string either `element` or `text`.

Parameter: **graph**
Return Type: **string**

getNodeValue

Get the value of this node if it is a text node, or the empty string if it is an element node

Parameter: **graph**
Return Type: **string**

getNumberOfChildren

Get the number of children of this node.

Parameter: **graph**
Return Type: **int**

getParentHCR

Get the HCR which contains this node Returns the graph representing the HCR or nilview(HCR).

Parameter: **graph**
Return Type: **graph**

getParentNode

Return the parent of this node.

Parameter: **graph** node
Return Type: **graph**

getPreviousSibling

The node immediately preceding this node in the list of its parent's children, or an empty graph if not found.

Parameter: **graph**
Return Type: **graph**

getValueFromHyperlink

Get the value referenced by a hyperlink. Returns the value or any(-1) if the value isn't found.

Parameter: **graph**
Return Type: **any**

getXMLFromGraph

Convert graph to a string of XML. Returns a string of XML.

Parameter: **graph**
Return Type: **string**

hasAttributes

Returns true if this node is an element type and it has some attributes.

Parameter: **graph**
Return Type: **bool**

hasChildNodes

Returns true if this node has any children.

Parameter: **graph**
Return Type: **bool**

insertBefore

Inserts the new node before the existing child node in the list of its parent's children. The first parameter is the new child node. The second parameter is the existing child node. Returns the parent graph with the new child inserted or an empty graph if the original child is not found.

Parameter: **graph**
Parameter: **graph**
Return Type: **graph**

isEqualNode

Compares two graphs to see if they represent the same program. This works by comparing the XML representations returned by *getXMLFromGraph*. The first and second parameters are the graphs to be compared. Returns true if the XML representations are the same.

Parameter: **graph**
Parameter: **graph**
Return Type: **boolean**

isSameNode

Return true if the two graph elements reference the same node value.

Parameter: **graph**
Parameter: **graph**
Return Type: **bool**

makeGraph

Construct a graph representation of a Hypercode program from an XML representation. The parameter is a string of XML. Returns the graph representation. Raises an error if it encounters mismatching XML tags.

Parameter: **string**
Return Type: **graph**

newElement

Creates a new graph node with a name and a list of attributes. The first parameter is the name of the graph node. The second parameter is the attributeList. Returns the new graph node.

Parameter: **string**
Parameter: **attributeList**
Return Type: **graph**

newHCRElement

Create a new HCR element. Returns a graph representing a new HCR element.

Return Type: **graph**

newHyperlinkElement

Create a new hyperlink element. The first parameter is the value which will be referenced by the hyperlink. The second parameter is the name of the hyperlink. The third parameter is the graph representing the HCR to which the hyperlink will belong. Returns a graph representing the hyperlink.

Parameter: **any**
Parameter: **string**
Parameter: **graph**
Return Type: **graph**

newTextElement

Creates a new graph node which only contains text. The parameter is the text contents of the node. Returns the new graph node.

Parameter: **string**
Return Type: **graph**

pasteHyperlink

Paste a copied hyperlink into a new graph as a child of the given graph node. The first parameter is the hyperlink node. The second parameter is the node to which the hyperlink will be attached as a child.

Parameter: **graph**
Parameter: **graph**

removeChild

Removes the first instance of this graph element in the list of children of its parent.

Parameter: **graph**

replaceChild

Replaces the first instance of the old graph element, in the list of its parent's children, with the new one. The first parameter is the element to be replaced. The second parameter is the new child element.

Parameter: **graph**
Parameter **graph**

setAttributeValue

Set the value of an attribute of the graph element, only applicable to element nodes. If the attribute does not yet exist, then add it. The first parameter is the name of the attribute. The second parameter is the new value of the attribute. The third parameter is the graph whore attribute will be set.

Parameter: **string**
Parameter: **string**
Parameter: **graph**

setNodeValue

Set the value of this graph element, only applicable to text nodes. The first parameter is a string (the text value). The second parameter is the graph element.

Parameter: **string**
Parameter: **graph**

updateHyperlinkLocation

Update a location referenced by a hyperlink. The first parameter is a graph representing the hyperlink which is to be updated - it must reference a location. The second parameter is a graph representing the value which will be placed in the location - must be the correct type for the location.

Parameter: **graph**
Parameter: **graph**

wrapHyperlinkInHCR

Wrap the graph in its parent HCR. Returns a graph representing the hyperlink in its HCR.

Parameter: **graph**
Return Type: **graph**

B.1.3 Evolution Pattern Operations

decomposeGraph

Decompose a composed behaviour, return the decompose sequence represented as a graph and a list of the connections, which the decompose has disconnected. These two values are used by *newComposeReplaceGraph* to recompose a new system. The parameter is a graph representing the code to decompose a value (which can be created by *newDecomposeGraph*). Returns a view with parts: *decompose_sequence*, which is a graph representing the sequence returned by the decompose operator; and *str4List*, which is a list containing information about which connections were connected in the decompose.

Parameter: **graph**
Return Type: **view[decompose_sequence:graph;saved_connections:str4List]**

getFirstBehaviourFromDecompose

Get a handle to the first behaviour in a decompose sequence. Returns a graph which represents a hyperlink to the first behaviour in the decompose sequence.

Parameter: **graph**
Return Type: **graph**

getFirstBehaviourRepresentationFromDecompose

Get a graph representation of the first behaviour in a decompose sequence. Returns a graph which represents a hyperlink to the first behaviour in the decompose sequence.

Parameter: **graph**
Return Type: **graph**

newBehaviourGraph

Create a graph representing an empty behaviour, which does nothing. Returns a graph representing an empty behaviour.

Return Type: **graph**

newComposeReplaceGraph

Construct a graph representing the compose statement to replace one behaviour in a composition with a new behaviour. The first parameter is a graph representing the

sequence returned by the decompose operator (obtained from *decomposeGraph*). The second parameter is a list containing information about which connections were connected in the decompose (obtained from *decomposeGraph*). The third parameter is a graph containing a hyperlink referencing the new behaviour for the composition. The fourth parameter is the name of the behaviour that is being replaced. Returns a graph representing a new compose statement.

Parameter: **graph**
Parameter: **str4List**
Parameter: **graph**
Parameter: **string**
Return Type: **graph**

newDeclarationGraphForGraph

Create a graph representing the declaration of a given value, represented as a graph. The first parameter is the name of the value to be declared. The second parameter is a graph representing the value. Returns a graph representing a declaration of the new value. Parameter: **string**
Parameter: **graph**
Return Type: **graph**

newDecomposeGraph

Create a graph representing the decompose clause for the given value, which must be a composed behaviour. The parameter is a composed behaviour. Returns a graph representing a decompose statement for this behaviour.

Parameters: **any**
Return Type: **graph**

Appendix C

Evolution Example Code

This appendix contains the code for the example presented in Chapter 6.

```
! Construct a graph representing the decomposition.
value decomposeGraph = newDecomposeGraph('System1)

! Evaluate the decomposition graph.
value decomposeResult = decomposeGraph(decomposeGraph)

! Get the sequence returned by the decompose statement.
value decomposeSequence = decomposeResult.decompose_sequence

value ProducerHyperlink = getFirstBehaviourFromDecompose(decomposeSequence)

value ProducerGraph = getGraphFromHyperlink(ProducerHyperlink)

value buildNewWidgetString = " .... "
value compilationResult = compile(buildNewWidgetString)

value buildNewWidget = project compilationResult.result as X onto
function[]→any: X() ! The result of X() is the buildNewWidget function.
default: any("")

value buildNewWidgetHyperlink = newHLElement(buildNewWidget,"buildNewWidget",Producer)

value hyperlinkList = getAllHyperlinks(ProducerGraph)

value list = location(hyperlinkList)   ! variable to store the search result

! search the list until a hyperlink with the name buildWidget is found
while 'list.node.name <> "buildWidget" do list := '('list.next)
value buildWidgetHyperlink = 'list.node  ! assign search result

replaceChild(buildWidgetHyperlink,buildNewWidgetHyperlink)

value Producer1Graph = newDeclarationGraphForGraph("Producer1",ProducerGraph)

value Producer1 = evaluateGraph(Producer1Graph)

value composeGraph = newComposeReplaceGraph(decomposeResult.decompose_sequence,
                          decompose_sequence.saved_connections, Producer1, "P")

value System1a = evaluateGraph(composeGraph)

updateHyperlinkLocation(System1,evolvedSystem)
```

Appendix D

Framework Evolution

This appendix gives a coded example of a meta-program that updates one of the MPF operations in the store.

The example in this appendix shows a meta-program that updates the printGraph operation. First the ArchWare ADL definition of the printGraph function is shown. It converts a Hypercode graph to a string. The meta-program will define a function called capitalize and add a call to it inside the declaration of tagName (near the end of printGraph).

```
recursive value printGraph = function(g: graphADL) -> string
{
    if not(g = nilview(graphADL))
        and not('(g.node) = nilview(tagTypeADL)) then
    {
        value node = '(g.node)
        if node.name = textTag then
        {
            node.text
        } else {

        ! printChildren .....
            value children = location("")
            if '(g.children) = nilview(graphADLList) then
                children := ""
            else {
                value list = location('(g.children))
                value result = location("")
                while not('list = nilview(graphADLList)) do
                {
                    value temp = printGraph('list.node)
                    result := 'result ++ temp
                    list := '('list.next)
                }
                children := 'result
            }

            value tagName = node.name
            "<" ++ tagName ++ printAttributes('(node.attributes)) ++ ">"
                    ++ 'children ++ "</"++ tagName ++">"
        }
    } else ""
}
```

The following code defines the meta-program that:

1. Extracts `printGraph` from the store.
2. Defines a function `capitalize`.
3. Alters the Hypercode graph of `printGraph` to include a call to `capitalize`.
4. Evaluates the new graph.
5. Replaces the old `printGraph` value in the store with the new one.

```
! Get the printGraph function value from the persistent store
value printGraphVal = googleADL("printGraph")

! Get the graph representation of the printGraph value
value printGraphRep = getGraphFromValue(printGraphVal)

! Define a function to capitalize the XML tags as a graph is printed out
value fun_def = "
value capitalize = function(str: string) -> string
{
        value upper = function(ch: string) -> string
        {
                if ch= '"a'" then '"A'"
                else if ch= '"a'" then '"A'"
                else if ch= '"b'" then '"B'"
                else if ch= '"c'" then '"C'"
                else if ch= '"d'" then '"D'"
                else if ch= '"e'" then '"E'"
                else if ch= '"f'" then '"F'"
                else if ch= '"g'" then '"G'"
                else if ch= '"h'" then '"H'"
                else if ch= '"i'" then '"I'"
                else if ch= '"j'" then '"J'"
                else if ch= '"k'" then '"K'"
                else if ch= '"l'" then '"L'"
                else if ch= '"m'" then '"M'"
                else if ch= '"n'" then '"N'"
                else if ch= '"o'" then '"O'"
                else if ch= '"p'" then '"P'"
                else if ch= '"q'" then '"Q'"
                else if ch= '"r'" then '"R'"
                else if ch= '"s'" then '"S'"
                else if ch= '"t'" then '"T'"
```

```
                else if ch= '"u'" then '"U'"
                else if ch= '"v'" then '"V'"
                else if ch= '"w'" then '"W'"
                else if ch= '"x'" then '"X'"
                else if ch= '"y'" then '"Y'"
                else if ch= '"z'" then '"Z'"
                else ch
        }

        value return = location('"'")
        value counter = location(1)
        while ''counter <= length(str) do {
                value ch = str(''counter|1)
                return := ''return ++ upper(ch)
                counter := ''counter + 1
        }
        ''return
}
capitalize
"

! Compile the capitalize function
value compresult = compile(fun_def, "trace")

! Project the result of compilation onto a function type
value capitalize = project compresult.result as X onto
function[]-> any      :{
        value x = X()   ! x is the any from the function
        x
}
default: {
        any("")
}

! Create a hyperlink referencing the new capitalize function
value capitalizeHL = newHLElement(capitalize,"capitalize",printGraphRep)

! Locate the correct place in the printGraph function to capitalize the tagname
! which is the last value declaration (see code for printGraph)
value valuedeclarationNodes = getAllNodesWithName("valuedeclaration",printGraphRep)

value temp = location(valuedeclarationNodes)
```

```
while not ('('temp.next) = nilview(graphADLList) ) do {
        temp := '('temp.next)
}

! Found the last value declaration
value lastValuedeclaration = 'temp.node

! Get the first clause node from the value declaration
value clauseNode = getFirstChildWithName("clause",lastValuedeclaration)

! Add a call to capitalize inside the clause
value closebracket  = newTextElement(")")
value openbracket = newTextElement("(")
value temp0 = appendChild(closebracket,clauseNode)
value temp1 = appendChildAt(openbracket,clauseNode,0)
value temp2 = appendChildAt(capitalizeHL,clauseNode,0)

! Create a recursice declaration for the new printGraph function
value recprint = newRecDeclarationGraphForGraph("printGraph",printGraphRep)

! Evaluate the graph
value printGraphHL = evaluateGraph(recprint)

! Get the new printGraph value so it can be put in the store
value newPrintGraphVal = getValueFromHyperlink(printGraphHL)

! Put the new function into the persistent store and delete the old printGraph function
value root = PS ()
project 'root as X onto ! root := view(system <- myEnv, users <- any(0))
view[system: any, users: any] : {
        project X.users as myEnv onto
        envADL: {
                myEnv.delete("printGraph")
                value temp3 = myEnv.put("printGraph",newPrintGraphVal)
        }
        default :{ }
}
default :{ }
abortcheckpoint()
```

Appendix E

Tower Model

This appendix sets the MPF in context by describing the Tower Model, an environment for developing evolving processes, which uses Hypercode as a mechanism for evolution.

The evolution enabled by the MPF should be under the control of a change management policy to ensure that any changes both improve the system and are applied in a non-disruptive manner. Change management is not part of the MPF, but it has been implemented as part of the ArchWare environment, which provides a hierarchical framework to structure the evolution[1].

E.1 Evolution in ArchWare

As part of the ArchWare project a change management system called the Tower Model (Greenwood et al., 2000) has been developed, which uses P2E technology to structure evolution. P2E is the Process for Process Evolution (Warboys et al., 1999a), a generic process for component based evolution.

In ArchWare, programs are developed in the ArchWare Environment. The Tower represents this Environment as a directed acyclic graph. Each node of the graph is an ArchWare Component, an evolvable subsystem that uses P2E to manage its own evolution. Components consist of an *evolver / producer* pair.

The Tower Model is designed to deal with change in large, complex and long-lived systems. To address the complexity, systems are designed heirachically and the components can be partitioned, or decomposed, into their sub-components. ArchWare Components in the Tower are associated with a set of process for managing change. These processes are themselves ArchWare Components, which are defined as follows:

Partition Partition an ArchWare Component to give its sub-components using decomposition.

Refine Refine an ArchWare Component to give its concrete representation, i.e., provide more detail on how the component will be implemented.

Satisfy Invoke satisfy on an ArchWare Component to determine whether the component satisfies a set of constraints. As a Component evolves, it may no longer satisfy its constraints, in which case either the Component must be evolved further or the constraints changed.

[1] Thanks to Mark Greenwood for his comments on this appendix.

Figure E.1 shows how nodes in an ArchWare Environment are made up of ArchWare Components, including the components which perform the *partition*, *satisfy* and *refine* operations. Inside, each component is defined as an *evolver* / *producer* pair.

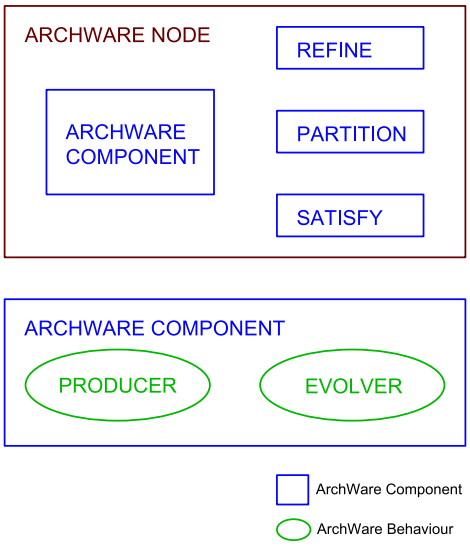


Figure E.1. ArchWare Envrionment. The ArchWare environment is a graph where each node is a set of ArchWare Components and each ArchWare Component is an *evolver* / *producer* pair.

To explain the application of the ArchWare environment model to a program, consider the example of a compiler in Figure E.2. The compiler has to *satisfy* a set of compilation rules. It is *partitioned* into a lexical analyser, type checker, code generator etc. *Refining* the compiler gives a handle to the executable compiler.

Structuring ArchWare Components as *evolver* / *producer* pairs allows the system to acommodate incremental evolution. The change process is encapsulated within Components, which can manage their own evolution. In an *evolver* / *producer* pair, the *producer* is the process which contains the application behaviour, and the *evolver* manages change to the *producer*. To understand how evolution is achieved, consider Figure E.3, which depicts an *evolver* and a *producer*. Change is initiated by *Managing*, which may be responding to an external signal or feedback from the *producer*. *Realise* applies the change. It has a handle to the *producer*, which allows it to decompose the *producer* into its constituent parts. *Technology* produces new elements by creating new values or making changes to existing values. It returns a set of abstractions which are composed by *Realise* to give a new behaviour, which is installed as the new *producer*.

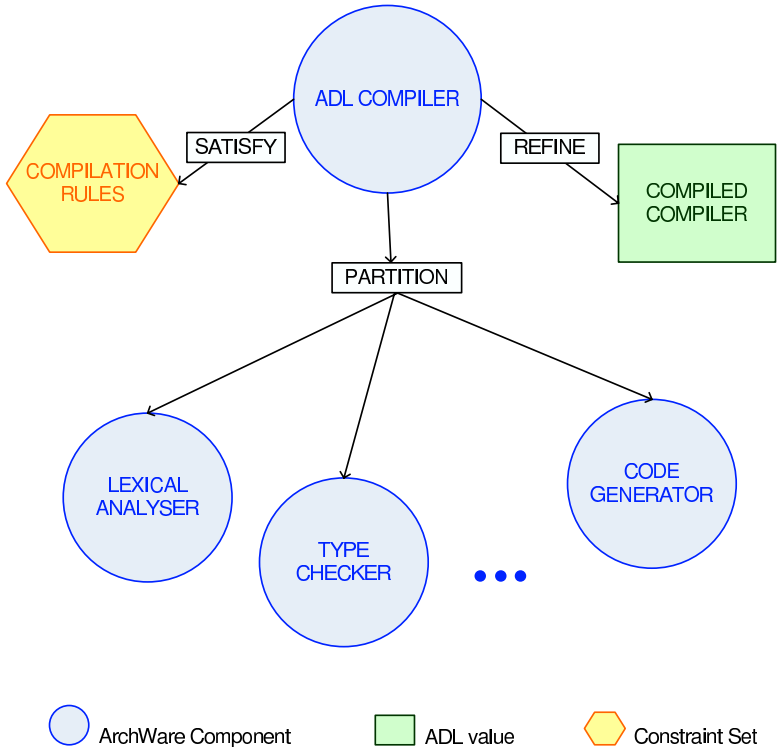


Figure E.2. ArchWare Components. Example of satisfy, partition and refine operating on an ArchWare Component.

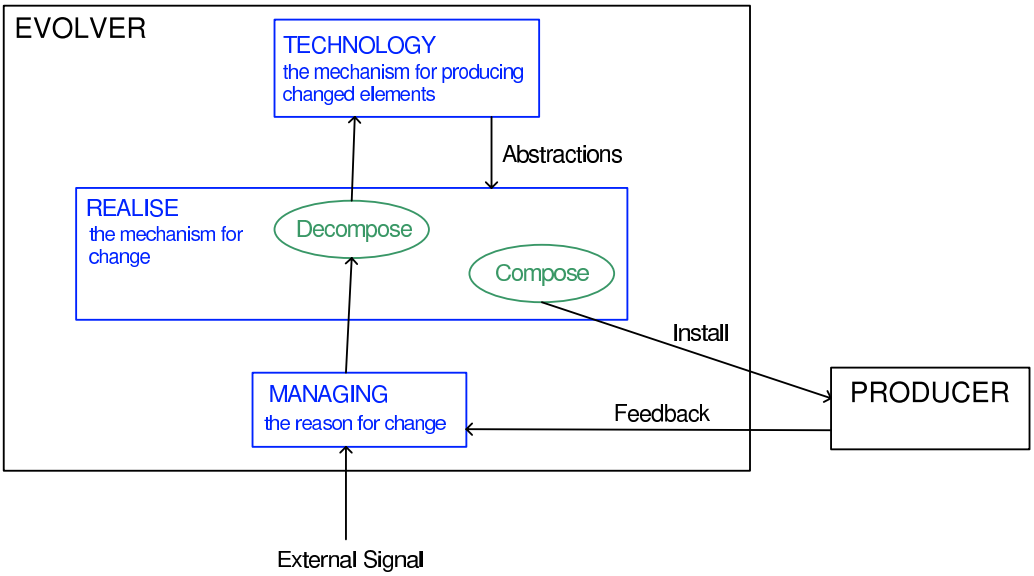


Figure E.3. Evolver / Producer. An Evolver component operates on a Producer to evolve it in response to some stimuli.

Bibliography

AHO-A. V., SETHI-R. AND ULLMAN-J. D. (1986). *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., ISBN 0-201-10088-6, Boston, MA, USA.

ALBANO-A., CARDELLI-L. AND ORSINI-R. (1985). Galileo: a strongly-typed, interactive conceptual language, *ACM Transactions on Database Systems*, **10**(2), pp. 230–260.

ALDRICH-J. (2003). *Using Types to Enforce Architectural Structure*, PhD thesis, University of Washington.

ALDRICH-J. (2005 (Submitted for publication)). Using types to enforce architectural structure, http://archjava.fluid.cs.cmu.edu/papers/archjava-overview.pdf.

ALDRICH-J., CHAMBERS-C. AND NOTKIN-D. (2002a). Architectural Reasoning in ArchJava, *Proceedings of the 16th European Conference on Object-Oriented Programming*, Springer-Verlag, Malaga, Spain, pp. 334–367.

ALDRICH-J., CHAMBERS-C. AND NOTKIN-D. (2002b). ArchJava: Connecting Software Architecture to Implementation, *Proceedings of the 24th International Conference on Software engineering*, ACM Press, Orlando, Florida, pp. 187–197.

ALDRICH-J., SAZAWAL-V., CHAMBERS-C. AND NOTKIN-D. (2002c). Architecture-Centric Programming for Adaptive Systems, *Proceedings of the 1st Workshop on Self-Healing Systems*, ACM Press, Charleston, South Carolina, pp. 93–95.

ALDRICH-J., SAZAWAL-V., CHAMBERS-C. AND NOTKIN-D. (2003). Language Support for Connector Abstractions, *European Conference on Object-Oriented Programming (ECOOP '03)*, Vol. 2743 of *Lecure Notes on Computer Science*, Springer, Darmstadt, Germany.

ALLEN-R. AND GARLAN-D. (1997). A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology*, **6**(3), pp. 213–249.

ALLEN-R., DOUENCE-R. AND GARLAN-D. (1998). Specifying and Analyzing Dynamic Software Architectures, *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lecture Notes in Computer Science 1382, Springer, Lisbon, Portugal.

ALLEN-R. J. (1997). *A Formal Approach to Software Architecture*, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

ANDERSON-J. R., CORBETT-A. T. AND REISER-B. J. (1986). *Essential LISP*, Addison-Wesley Longman Publishing Co., Inc., ISBN 0-201-11148-9.

ATKINSON-M. AND MORRISON-R. (1985). Procedures as Persistent Data Objects, *ACM Transactions on Programming Languages and Systems*, **7**(4), pp. 539–559.

ATKINSON-M. AND MORRISON-R. (1995). Orthogonally Persistent Object Systems, *VLDB Journal*, **4**(3), pp. 319–401.

ATKINSON,-M. AND WELLAND,-R. (EDS.) (2000). *Fully Integrated Data Environments: Persistent Programming Languages, Object Stores, and Programming Environments*, Esprit Basic Research Series, Springer Verlag, Germany, ISBN 354065772X.

ATKINSON-M., BAILEY-P., CHISHOLM-K., COCKSHOTT-W. AND MORRISON-R. (1983). An Approach to Persistent Programming, *Computer Journal*, **26**(4), pp. 360–365.

ATKINSON-M., CHISHOLM-K. AND COCKSHOTT-P. (1982). PS-algol: An Algol with a Persistent Heap, *ACM SIGPLAN Notices*, **17**(7), pp. 24–31.

ATKINSON-M., DAYNES-L., JORDAN-M., PRINTEZIS-T. AND SPENCE-S. (1996a). An orthogonally persistent Java, *ACM SIGMOD Record*, **25**(4), pp. 68–75.

ATKINSON-M. P., JORDAN-M. J., DAYNÈS-L. AND SPENCE-S. (1996b). Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system, *Proceedings of the 7th Workshop on Persistent Object Systems (POS'96), Cape May (NJ), USA*, pp. 33–47.

BALASUBRAMANIAM-D., MORRISON-R., KIRBY-G., MICKAN-K. AND NORCROSS-S. (2004a). ArchWare ADL Release 1 User Reference Manual, *Technical Report D4.3*, ArchWare Project IST-2001-32360.

BALASUBRAMANIAM-D., MORRISON-R., KIRBY-G., MICKAN-K., WARBOYS-B., ROBERTSON-I., SNOWDON-B., GREENWOOD-M. AND SEET-W. (2005). A software architecture approach for structuring autonomic systems, *ICSE 2005 Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005), St Louis, MO, USA*, ACM Digital Library, ISBN 1-59593-963-2, pp. 59–65.

BALASUBRAMANIAM-D., MORRISON-R., MICKAN-K., KIRBY-G., WARBOYS-B., ROBERTSON-I., B.SNOWDON, GREENWOOD-R. AND SEET-W. (2004b). Support for feedback and change in self-adaptive systems, *ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04), Newport Beach, CA, USA*, ACM Press, New York, NY, USA, ISBN 1-58113-989-6, pp. 18–22.

BROOKS-F. P. (1987). No silver bullet: Essence and accidents of software engineering, *Computer, IEEE Computer Society Press*, **20**(4), pp. 10–19.

BROSILOW-C. AND JOSEPH-B. (2002). *Techniques of Model-Based Control*, Prentice-Hall, chapter Chapter 9: Feedforward Control.

BUCKLEY-J., MENS-T., ZENGER-M., RASHID-A. AND KNIESEL-G. (2005). Towards a taxonomy of software change, *Journal on Software Maintenance and Software Evolution: Research and Practice (Wiley) - to appear*.

BURSTALL-R., COLLINS-J. AND POPPLESTONE-R. (1971). *Programming in POP-2*, Edinburgh University Press, Edinburgh, Scotland, ISBN 0852241976.

CHENG-S.-W., HUANG-A.-C., GARLAN-D., SCHMERL-B. AND STEENKISTE-P. (2004). An Architecture for Coordinating Multiple Self-Management Systems, *in* J. Magee, C. Szyperski and J. Bosch (eds.), *4th Working IEEE/IFIP Conference on Software Architecture (WICSA 4)*, IEEE, Oslo, Norway, pp. 243 – 252.

CHESS-D. M., SEGAL-A., WHALLEY-I. AND WHITE-S. R. (2004). Unity: Experiences with a prototype autonomic computing system, *In the Proceedings of the International Conference on Autonomic Computing (ICAC'04)*, IEEE Computer Society, New York, NY, USA, pp. 140–147.

CUTTS-Q., CONNOR-R. AND MORRISON-R. (2000). *Fully Integrated Data Environments: Persistent Programming Languages, Object Stores, and Programming Environments*, Springer Verlag, chapter The PamCase Machine, pp. 346 – 364.

CZARNECKI-K. AND EISENECKER-U. (2000). *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, USA, ISBN 0201309777.

DEARLE-A. (1987). Constructing compilers in a persistent environment, *in* M. Atkinson, O. Buneman and R. Morrison (eds.), *2nd International Workshop on Persistent Object Systems, Appin, Scotland*, pp. 443 – 455.

DEARLE-A. AND BROWN-A. (1988). Safe browsing in a strongly typed persistent environment, *Computer Journal*, **31**(6), pp. 540–544.

DEUX-O. (1990). The Story of O2, *IEEE Transactions on Knowledge and Data Engineering*, **2**(1), pp. 91–108.

DEWITT-A., GROSS-T., LOWEKAMP-B., MILLER-N., STEENKISTE-P., SUBHLOK-J. AND SUTHERLAND-D. (1998). ReMoS: A resource monitoring system for network-aware applications, *Technical Report CMU-CS-97-194*, Carnegie Mellon School of Computer Science, Pittsburgh, PA.

DIAO-Y., HELLERSTEIN-J., PAREKH-S. AND BIGUS-J. (2003). Managing web server performance with AutoTune agents, *IBM Systems Journal*, **42**(1), pp. 136–149.

DMITRIEV-M. (2000). Class and Data Evolution Support in the PJama Persistent Platform, *Technical Report TR 2000-57*, Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, G12 8RZ, Scotland.

EBRAERT-P., MENS-T. AND D'HONDT-T. (2004). Enabling dynamic software evolution through automatic refactorings, *In the proceedings of the Workshop on Software Evolution Transformations (SET2004) in conjunction with the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, Delft, Netherlands.

ENDLER-M. AND WEI-J. (1992). Programming Generic Dynamic Reconfigurations for Distributed Applications, *Proceedings of the 1st International Workshop on Configurable Distributed Systems*, IEE, Imperial College, London, pp. 68–79.

FARKAS-A. AND DEARLE-A. (1994). The Octopus model and its Implementation, *Proceedings of the 17th Australiasian Computer Science Conference (ACSC)*, **16**(1), pp. 581–590.

FOSSA-H. AND SLOMAN-M. (1996). Implementing Interactive Configuration Management for Distributed Systems, *Proceedings of the 3rd International Conference on Configurable Distributed Systems (ICCDS '96)*, Annapolis MA, pp. 44–51.

GARLAN-D., ALLEN-R. AND OCKERBLOOM-J. (1994). Exploiting Style in Architectural Design Environments, *Proceedings of SIGSOFT'94: The 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, pp. 175–188.

GARLAN-D. AND PERRY-D. E. (1995). Introduction to the Special Issue on Software Architecture, *IEEE Transactions on Software Engineering*, **21**(4), pp. 269–274.

GARLAN-D. AND SCHMERL-B. (2002). Model-based adaptation for self-healing systems, *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*, ACM Press, New York, NY, USA, pp. 27–32.

GARLAN-D. AND SHAW-M. (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall Publishing, ISBN 0131829572.

GARLAN-D., CHENG-S.-W. AND SCHMERL-B. (2003). Increasing System Dependability through Architecture-based Self-repair, *in* R. de Lemos, C. Gacek and A. Romanovsky (eds.), *Architecting Dependable Systems*, Springer-Verlag, New York, ISBN 3540407278, pp. 61–90.

GARLAN-D., CHENG-S.-W., HUANG-A.-C., SCHMERL-B. AND STEENKISTE-P. (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure, *IEEE Computer*, **37**(10), pp. 46–54.

GARLAN-D., MONROE-R. T. AND WILE-D. (1997). ACME: An Architecture Description Interchange Language, *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'97)*, IBM Press, Toronto, Ontario, pp. 169–183.

GARLAN-D., SCHMERL-B., J. AND CHANG (2001). Using gauges for architecture-based monitoring and adaptation, *Proceedings of the Working Conference on Complex and Dynamic System Architecture*, Brisbane, Australia.

GEORGIADIS-I. (2002). *Self-Organising Distributed Component Software Architecture*, PhD thesis, Department of Computing, Imperial College, University of London, UK.

GEORGIADIS-I., MAGEE-J. AND KRAMER-J. (2002). Self-organising software architectures for distributed systems, *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*, ACM Press, New York, NY, USA, pp. 33–38.

GOLDBERG-A. AND ROBSON-D. (1983). *Smalltalk-80 - The language and its implementation*, Addison-Wesley, ISBN 0-201-11371-6.

GORLICK-M. M. AND RAZOUK-R. R. (1991). Using Weaves for Software Construction and Analysis, *Proceedings of the 13th International Conference on Software Engineering*, IEEE Computer Society Press, Austin, Texas, United States, pp. 23–34.

GREENWOOD-M., ROBERTSON-I. AND WARBOYS-B. (2000). A support framework for dynamic organizations, *in* R. Conradi (ed.), *Proceedings of the 7th European Workshop in Software Process Technology (EWSPT 2000)*, Vol. 1780 of *Lecture Notes in Computer Science*, Springer, Kaprun, Austria, pp. 6–20.

GREENWOOD-R., BALASUBRAMANIAM-D., CÍMPAN-S., KIRBY-G., MICKAN-K., MORRISON-R., OQUENDO-F., ROBERTSON-I., SEET-W., SNOWDON-B., WARBOYS-B. AND ZIRINTSIS-E. (2003). Process support for evolving active architectures, *in* F. Oquendo (ed.), *Proceedings of the 9th European Workshop on Software Process Technology (EWSPT 2003)*, Lecture Notes in Computer Science 2786, Springer-Verlag, ISBN 3-540-40764-2, Helsinki, Finland, pp. 112–127.

HEINEMAN-G. T. (1997). A model for designing adaptable software components, *Technical Report WPI-CS-TR-97-6*, Worcester Polytechnic Institute, Computer Science Department, Worcester, Massachusetts.

HOARE-C. (1985). *Communicating Sequential Processes*, Prentice-Hall International Series in Computing Science, Prentice-Hall International, Englewood Cliffs, N.J., ISBN 0131532715.

HOFMEISTER-C. R. (1993). *Dynamic Reconfiguration of Distributed Applications*, PhD thesis, Computer Science Department, University of Maryland.

HOOK-J. AND SHEARD-T. (1993). A semantics of compile time reflection, *Technical Report 93-019*, Dept of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, Portland, Oregon.

HOSKING-A. AND NOVIANTO-A. (1997). Reachability-based Orthogonal Persistence for C, C++ and Other Intransigents, *in* P. Dickman and P. R. Wilson (eds.), *Proceedings of the OOPSLA Workshop on Memory Management and Garbage Collection*, Atlanta, Georgia.

IEE (2000). *IEEE Std 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems*.

JACKSON-D. (1999). Alloy: A Lightweight Object Modelling Notation, *Software Engineering and Methodology*, **11**(2), pp. 256–290.

KEPHART-J. O. (2005). Research challenges of autonomic computing, *ICSE '05: Proceedings of the 27th international conference on Software engineering*, ACM Press, New York, NY, USA, pp. 15–22.

KEPHART-J. O. AND CHESS-D. M. (2003). The Vision of Autonomic Computing, *Computer, IEEE Computer Society Press*, **36**(1), pp. 41–50.

KICZALES-G., HILSDALE-E., HUGUNIN-J., KERSTEN-M., PALM-J. AND GRISWOLD-W. (2001). Getting started with AspectJ, *Communications of the ACM*, **44**(10), pp. 59–65.

KICZALES-G., LAMPING-J., MAEDA-C., KEPPEL-D. AND MCNAMEE-D. (1993). The need for customizable operating systems, *Proceedings of the Fourth Workshop on Workstation Operating Systems*, IEEE Computer Society Technical Committee on Operating Systems and Applications Environment, IEEE Computer Society Press, pp. 165–169.

KIRBY-G. (1992). Persistent Programming with Strongly Typed Linguistic Reflection, *in* R. Morrison and M. Atkinson (eds.), *25th International Conference on Systems Sciences (HICSS)*, Vol. 2, Hawaii, pp. 820–831.

KIRBY-G. AND DEARLE-A. (1990). An adaptive graphical browser for napier88, *Technical Report CS/90/16*, University of St Andrews.

KIRBY-G. AND MORRISON-R. (1997). Ocb: An object/class browser for Java, *in* M. Jordan and M. Atkinson (eds.), *2nd International Workshop on Persistence and Java (PJW2), Half Moon Bay, California*, pp. 89–105.

KIRBY-G., CONNOR-R., CUTTS-Q., DEARLE-A., FARKAS-A. AND MORRISON-R. (1992). Persistent Hyper-Programs, *5th International Workshop on Persistent Object Systems*, Springer-Verlag, San Miniato, Italy, pp. 86–106.

KIRBY-G., CUTTS-Q., CONNOR-R. AND MORRISON-R. (1993). The Implementation of a Hyper-Programming System, *Technical Report CS/93/5*, University of St Andrews.

KIRBY-G., MORRISON-R. AND STEMPLE-D. (1998). Linguistic Reflection in Java, *Software - Practice and Experience*, **28**(10), pp. 1045–1077.

KRAMER-J. AND MAGEE-J. (1990). The Evolving Philosophers Problem: Dynamic Change Management, *IEEE Transactions on Software Engineering*, **16**(11), pp. 1293–1306.

LEHMAN-M. M. (1996). Laws of software evolution revisited, *in* C. Montangero (ed.), *5th European Workshop on Software Process Technology, EWSPT*, Lecture Notes in Computer Science 1149, Springer, Nancy, France, pp. 108–124.

MAGEE-J. AND KRAMER-J. (1996a). Dynamic Structure in Software Architectures, *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ACM Press, San Francisco, California, United States, pp. 3–14.

MAGEE-J. AND KRAMER-J. (1996b). Self Organising Software Architectures, *Joint Proceedings of the 2nd International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, ACM Press, San Francisco, California, United States, pp. 35–38.

MAGEE-J., DULAY-N. AND KRAMER-J. (1994). Regis: a constructive development environment for distributed programs, *Distributed Systems Engineering Journal: Special Issue on Configurable Systems*, **1**(5), pp. 304–312.

MAGEE-J., DULAY-N., EISENBACH-S. AND KRAMER-J. (1995). Specifying Distributed Software Architectures, *Proceedings of the 5th European Software Engineering Conference*, Springer-Verlag, Sitges, Spain, pp. 137–153.

MCCABE-T. J. AND WATSON-A. H. (1994). Software complexity, *Crosstalk, Journal of Defense Software Engineering*.

MEDVIDOVIC-N. (1996). ADLs and Dynamic Architecture Changes, *Joint Proceedings of the 2nd International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, ACM Press, San Francisco, California, United States, pp. 24–27.

MENS-T., BUCKLEY-J., ZENGER-M. AND RASHID-A. (2003). Towards a taxonomy of software evolution, *Workshop on Unanticipated Software Evolution*, Warshau, Poland.

MICKAN-K., MORRISON-R., KIRBY-G., BALASUBRAMANIAM-D. AND ZIRINTSIS-E. (2004). Using generative programming to visualise Hypercode in complex and dynamic systems, *Proceedings of the 27th Australasian Computer Science Conference (ACSC2004)*, Australian Computer Society, Inc., ISBN 1-920682-05-8, Dunedin, New Zealand, pp. 377–386.

MILNER-R. (1999). *Communicating and Mobile Systems: The* Π*-Calculus*, Cambridge University Press, ISBN 0521658691.

MONROE-R. T. (1998). Capturing Software Architecture Design Expertise with Armani, *Technical Report TR CMU-CS-98-163*, Carnegie Mellon University School of Computer Science.

MORRISON-R., BALASUBRAMANIAM-D., GREENWOOD-M., KIRBY-G., MAYES-K., MUNRO-D. AND WARBOYS-B. (1999a). ProcessBase Reference Manual (Version 1.0.6), *Technical report*, Universities of St Andrews and Manchester.

MORRISON-R., BALASUBRAMANIAM-D., GREENWOOD-R., KIRBY-G., MAYES-K., MUNRO-D. AND WARBOYS-B. (2000a). An Approach to Compliance in Software Architectures, *IEE Computing and Control Engineering Journal, Special Issue on Informatics*, **11**(4), pp. 195–200.

MORRISON-R., BROWN-A., CARRICK-R., CONNOR-R., DEARLE-A. AND M.P. ATKINSON-M. (1990). The Napier Type System, *in* J. Rosenberg and D. Koch (eds.), *Persistent Object Systems*, Springer-Verlag, pp. 3–18.

MORRISON-R., CONNOR-R., CUTTS-Q., DEARLE-A., FARKAS-A., KIRBY-G., MCGETTRICK-R. AND ZIRINTSIS-E. (1999b). Current directions in hyper-programming, *Lecture Notes in Computer Science 1755*, Springer-Verlag, pp. 316–340.

MORRISON-R., CONNOR-R., KIRBY-G. AND MUNRO-D. (1996). Can Java persist?, *Proceedings of the 1st International Workshop on Persistence for Java (PJW1), Drymen, Scotland*.

MORRISON-R., CONNOR-R., KIRBY-G., MUNRO-D., ATKINSON-M., CUTTS-Q., BROWN-A. AND DEARLE-A. (2000b). The Napier88 Persistent Programming Language and Environment, *in* M. Atkinson and R. Welland (eds.), *Fully Integrated Data Environments: Persistent Programming Languages, Object Stores, and Programming Environments*, Springer-Verlag, Europe, pp. 98–154.

MORRISON-R., KIRBY-G., BALASUBRAMANIAM-D., MICKAN-K., OQUENDO-F., CÍMPAN-S., WARBOYS-B., SNOWDON-B. AND GREENWOOD-R. (2003). Constructing Active Architectures in the ArchWare ADL, *Technical Report CS/03/3*, University of St Andrews.

MORRISON-R., KIRBY-G., BALASUBRAMANIAM-D., MICKAN-K., OQUENDO-F., CÍMPAN-S., WARBOYS-B., SNOWDON-B. AND GREENWOOD-R. (2004). Support for evolving software architectures in the ArchWare ADL, *in* J. Magee, C. Szyperski and J. Bosch (eds.), *4th Working IEEE/IFIP Conference on Software Architecture (WICSA 4)*, IEEE, ISBN 0-7695-2172-X, Oslo, Norway, pp. 69 – 78.

OQUENDO-F., WARBOYS-B., MORRISON-R., DINDELEUX-R., GALLO-F., GARAVEL-H. AND OCCHIPINTI-C. (2004). ArchWare: Architecting Evolvable Software, *in* F. Oquendo, B. Warboys and R. Morrison (eds.), *Proceedings of the First European Workshop on Software Architecture, EWSA 2004*, Vol. 3047 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 257–271.

OREIZY-P. AND TAYLOR-R. (1998). On the Role of Software Architectures in Runtime System Reconfiguration, *Proceedings of the International Conference on Configurable Distributed Systems (ICCDS 4)*, IEEE Computer Society, Annapolis MD.

OREIZY-P., GORLICK-M. M., TAYLOR-R. N., JOHNSON-G., MEDVIDOVIC-N., QUILICI-A., ROSENBLUM-D. S., AND WOLF-A. L. (1999). An architecture-based approach to self-adaptive software, *IEEE Intelligent Systems*, **14**(3), pp. 54–62.

OREIZY-P., MEDVIDOVIC-N. AND TAYLOR-R. N. (1998). Architecture-Based Runtime Software Evolution, *Proceedings of the 20th International Conference on Software Engineering*, IEEE Computer Society, Kyoto, Japan, pp. 177–186.

RICHARDSON-J. E. AND CAREY-M. J. (1989). Persistence in the E language: Issues and implementation, *Software Practice and Experience*, **19**(12), pp. 1115–1150.

SCHMERL-B., ALDRICH-J., GARLAN-D., KAZMAN-R. AND YAN-H. (2005). Discovering architectures from running systems using colored petri nets, submitted for publication.

SCHMERL-B. AND GARLAN-D. (2002). Exploiting Architectural Design Knowledge to Support Self-repairing Systems, *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, ACM Press, Ischia, Italy, pp. 241–248.

SCHMIDT-J. W. AND MATTHES-F. (1994). The DBPL project: advances in modular database programming, *Information Systems*, **19**(2), pp. 121–140.

SHEARD-T. (1998). Using MetaML: A staged programming language, *Lecture Notes of the Summer School on Advanced Functional Programming*, **LNCS 1129**, pp. 207–239.

SIMONYI-C. (1995). The Death of Computer Languages, the Birth of Intentional Programming, *Technical Report MSR-TR-95-52*, Microsoft Research Microsoft Corporation.

SIMONYI-C. (1996). Intentional Programming: Innovation in the Legacy Age.

STEMPLE-D., FEGARAS-L., STANTON-R., SHEARD-T., PHILBROW-P., COOPER-R., ATKINSON-M., MORRISON-R., KIRBY-G., CONNOR-R. AND ALAGIC-S. (2000). *Fully Integrated Data Environments: Persistent Programming Languages, Object Stores, and Programming Environments*, Springer Verlag, Germany, chapter Type-Safe Linguistic Reflection: A Generator Technology, pp. 158–188.

STRACHEY-C. (1967). Fundamental concepts in programming languages, Oxford University Press. Republished in Higher Order Symbolic Computation (2000), 13(1-2), pp. 11– 49.

STRAW-A., MELLENDER-F. AND RIEGEL-S. (1989). Object Management in a Persistent Smalltalk System, *Software Practice Experience*, **19**(8), pp. 719–737.

TENNENT-R. D. (1977). Language Design Methods Based on Semantic Principles, *Acta Informatica*, **8**(2), pp. 97–112.

VAN WYK-E., DE MOOR-O., SITTAMPALAM-G., PIRETTI-I. S., BACKHOUSE-K. AND KWIATKOWSKI-P. (2001). Intentional Programming: a Host of Language Features, *Technical Report PRG-RR-01-21*, Oxford University Computing Laboratory: Programming Research Group, Oxford.

W3 CONSORTIUM (1999). XPath 1.0: XML path language. http://www.w3.org/TR/xpath/.

W3 CONSORTIUM (2004). XML Information Set. http://www.w3.org/TR/xml-infoset.

W3 CONSORTIUM (2005a). Document object model. http://www.w3.org/DOM/.

W3 CONSORTIUM (2005b). XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/.

WARBOYS-B., AVRILLIONIS-D., CONRADI-R., CUNIN-P.-Y., NGUYEN-M. N. AND ROBERTSON-I. (1999a). Meta-process, *Software Process: Principles, Methodology, Technology*, Vol. 1500 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 53–94.

WARBOYS-B., BALASUBRAMANIAM-D., GREENWOOD-R., KIRBY-G., MAYES-K., MORRISON-R. AND MUNRO-D. (1999b). Collaboration and Composition: Issues for a Second Generation Process Language, *Proceedings of the 7th European Software Engineering Conference (ESEC'99)*, Lecture Notes in Computer Science 1687, Springer-Verlag, pp. 75–91.

YAN-H., GARLAN-D., SCHMERL-B., ALDRICH-J. AND KAZMAN-R. (2004). Discotect: A system for discovering architectures from running systems, *Proceedings of the 26th International Conference on Software Engineering (ICSE'04) Edinburgh, Scotland*, IEEE Computer Society, Washington, DC, USA, pp. 470–479.

ZIRINTSIS-E. (2000). *Towards Simplification of the Software Development Process: The Hyper-Code Abstraction*, PhD, University of St Andrews.

ZIRINTSIS-E., KIRBY-G. AND MORRISON-R. (2001). Hyper-Code Revisited: Unifying Program Source, Executable and Data, *Lecture Notes in Computer Science 2135*, Springer, pp. 232–246.

Glossary

This is a list of commonly used terms and acronyms used in this thesis. The page numbers for each entry refer to the first use in the text.

ADL Architecture Description Language., 9

API Application Programming Interface. An interface that a program (or part of a program) provides in order to allow requests for service to be made of it by other programs (or parts of programs), and/or to allow data to be exchanged between them., 26

CFS Context Free Syntax. A formal syntax in which every production rule is of the form: $V \rightarrow w$, where V is a non-terminal symbol and w is a string consisting of terminals and/or non-terminals. It is context-free because the non-terminal V can always be replaced by w, regardless of the context in which it occurs., 200

Closed system A software system that uses information extracted from its own execution for self-adjustment., 5

Compose Connect a sequence of components and start their execution., 94

Constraints See Properties., 5

Decompose Stop part of an executing system, allowing the remaining parts to continue executing., 14

Framework A system, or set of components, that operates over another system., 13

Injection An injection operation takes values of type T and returns values of type C(T), where C is some type constructor., 80

Open system A software system that can incorporate environmental input during evolution., 4

Program closure The set of a program and all its values., 17

Projection Projection is the inverse of injection., 80

Properties Constraints on a software architecture, e.g. each client may be connected to only one server., 9

Referential Integrity Once a reference to an object has been established that object will remain accessible for the lifetime of the reference., 27

Software Architecture A system model at a high level of abstraction, commonly in terms of components and connectors., 9