UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Tool Coordination in METAFrame

A. Claßen, B. Steffen, T. Margaria, V. Braun

Lehrstuhl für Programmiersysteme
Universität Passau
Innstr. 33
D-94032 Passau (Germany)
tel: +49 851 509.3090
fax: +49 851 509.3092
{classen,tiziana,steffen,v.braun}@fmi.uni-passau.de

MIP-9707
April 1997

Abstract

The MetaFrame tool integration platform is designed for the interactive combination and coordination of heterogeneous tools: complex tool combinations can be (semi-) automatically or interactively constructed and tested by on-line 'meta-programming' in the interpreted, sequential tool coordination language HLL. The point of our design is the strict separation of a highly enhanced integration phase and a strongly supported coordination phase. This allows even non-experts to profitably coordinate the integrated tools to solve their own application-specific tasks. It is this profile which qualifies our coordination environment to serve as the Electronic Tool Integration (ETI) platform in the new international Springer Journal *Software Tools for Technology Transfer* (STTT).

Contents

1 Introduction

As "real–life" industrial problems become more and more complex, engineers will not be able to solve them using a single method or a single tool alone. The solution lies in resorting to the use of heterogeneous methods, which require the appropriate *integration and coordination of existing tools*. Combining tools allows us to make optimal use of their individual strengths and to avoid their weaknesses by using the most adequate tool for each task. This approach contrasts the frequent practice of extending one tool to make it universally applicable to a variety of tasks. Besides the development cost, this would increase the tool's complexity, making it very difficult to use and to maintain. It also would force the engineers to learn how to use the extended tool for tasks they were used to solve with familiar, simpler tools.

The paper describes the METAFrame open tool coordination platform, which allows the interactive combination and coordination of heterogeneous tools. Tool combinations can be automatically or interactively constructed, and tested by on-line 'meta-programming' in the interpreted sequential tool coordination language HLL. HLL control structures allow specifying even complex coordination tasks in an intuitive manner, while its data types, high-level data structures, and the elaborate METAFrame graphical user interface provide comfortable management and presentation of input and output data. Interactive experimenting with tool combinations is supported by METAFrame's hypertext system (for browsing documentation) and by an automatic tool synthesis mechanism that generates HLL code from loose, application-specific specifications.

METAFrame's coordination features have already been used in a large number of application areas (see Figure 3), including industrial projects. They form the basis for the Electronic Tool Integration (ETI) component, the electronic part of Springer's Software Tools for Technology Transfer (STTT) journal.[1] Its integration and coordination aspects are designed to support even unexperienced users or newcomers in the profitable coordination of integrated tools to solve their own application-specific tasks: the integration and the coordination phase are strictly separated, and the integration phase is strongly enhanced.

Tool integration: Tools are integrated into METAFrame through individual *tool adapters*, which capsulate a tool's behaviour and present a uniform interface to the coordination layers. Tool adapters are obtained either by direct C++ programming or, preferably, via an *adapter generator*, which constitutes our current automatic integration support. Requiring programming skills, as well as technical knowledge about the single tools, the application domain, tool adapters, data conversion and control passing, integration should (at least initially[2]) be taken care of by integration designers/managers with specific expert knowledge.

[1] See http://brahms.fmi.uni-passau.de/bs/sttt/ for details of this journal.

[2] We are working on additional (semi-) automatic integration support, opening also this phase to a wider community.

Tool adapters are compiled separately and loaded dynamically into the MetaFrame system. Thus new tools can be integrated into the running system without requiring a restart or even a recompilation of MetaFrame, which therefore provides an open coordination environment.

Tool coordination: On the contrary, we wish the use of already integrated tools to be as simple and as intuitive as possible, so that little or no specific knowledge is prerequisite to the use of MetaFrame as a coordination environment. In particular, high-level task specification languages, graphical support for specifications and user interaction, as well as automatic support of the coordination activity by means of automatic synthesis and prototype animation are offered.

Trade off: Since a tool's integration is required only once for each tool, yielding a tool-specific adapter, while tool coordination is the main purpose and is done over and over again, our choice of moving alarge part of the effort into the integration phase (resulting in a less technical and much more user friendly coordination representation of the integrated tools), pays off repeatedly for the users performing tool coordination. The strict separation of integration and coordination as well as the gains in ease of coordination resulting from a shift of effort towards higher integration costs constitutes a design principle of crucial importance for the practicality of MetaFrame. Both in the ETI and in our industrial projects, acceptance and usability of the environment, and consequently its success, are largely due to this principle. The integration effort spent by a small group of experts (at our site for the ETI, a small designer team at Siemens/Siemens Nixdorf) guarantees comfortable, non–technical tool coordination for the whole ETI user community and for all the customers of the INXpress Service Design environment [24] respectively.

While the following two sections are devoted to the coordination (Sect. 2) and integration (Sect. 3) respectively, Sect. 4 presents the MetaFrame coordination environment and Sect. 5 shows the impact of the achieved flexibility along concrete application examples. Finally, Sect. 7 discusses related approaches and Appendix A goes into more technical details along a specific example.

2 Tool Coordination

Coordination means the high–level specification, generation, and management of tool combinations. It spans from the specification of (sequences or patterns of) tool invocations to the management of tool data. MetaFrame offers a variety of coordination languages on different abstraction levels (see Figure 1), which are typically tailored for different user groups:

- Our basic coordination language is the interpreted, imperative language HLL (**H**igh–**L**evel **L**anguage), which provides

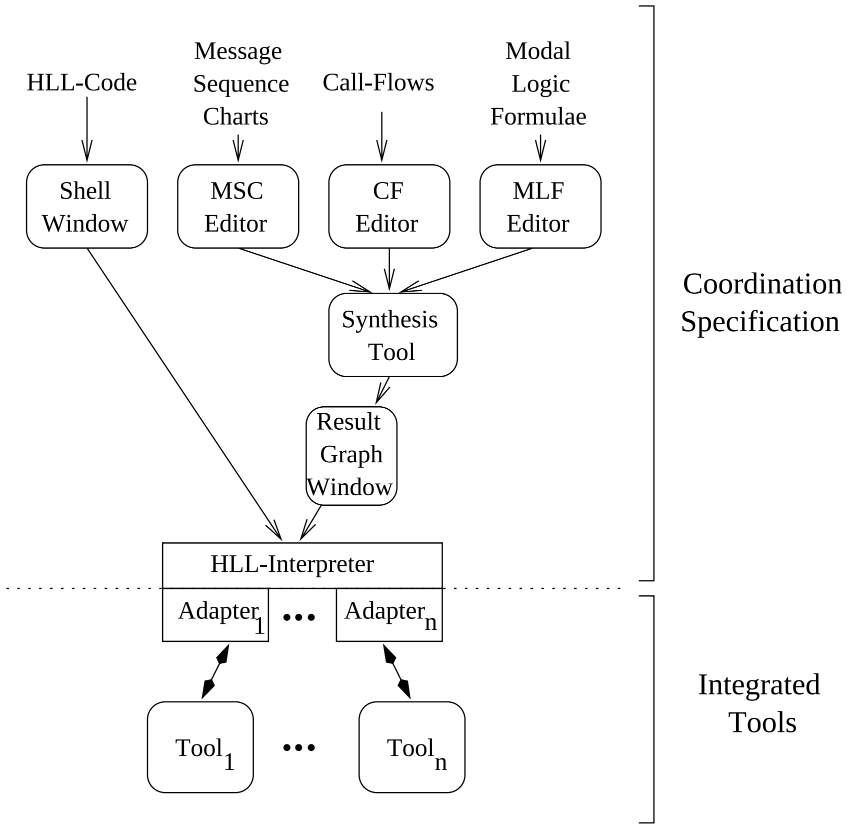

Figure 1: The MetaFrame Tool Coordination Environment

- a *uniform view* on the tools, abstracting from tool-specific technical details like data formats or invocation modalities. As will be detailed in Sect. 4.2, tool invocations are in fact simply represented by HLL function calls, while HLL data values are abstract representations for tool-specific data formats.
- a handy *programming language* for the combination of tools, at the adapter abstraction level.

- Three higher–level coordination formalisms are also supported by MetaFrame, in the spirit of a dual-layer approach to coordination specification [4]. In particular these formalisms aim at giving each user the possibility of specifying coordination tasks in the accustomed, application-dependent way: By means of graphical languages like *Message Sequence Charts* (a standard language in communication industry [14]) and *Call–Flows* (widely used in the specification of telecommunication services), as well as *Modal Logic Formulae* (which are e.g. familiar to the verification community), quite different groups of users are able to specify in a familiar way loose coordination requirements, which are far more abstract than any concrete HLL program.

Once entered into MetaFrame's language-specific editor windows, these requirements are transformed into concrete HLL coordination sequences by an *automatic synthesis* tool [26, 4, 27] (see Figure 1). Since these specification formalisms are tailored to express loose requirements, the synthesis process typically inserts intermediate transformation steps needed to guarantee consistent, executable solutions. The solution space resulting from the automatic synthesis may contain quite different alternatives, each corresponding to a different minimal-length sequence of tools satisfying the specified (loose) constraints. The potential of alternative solutions is presented as a graph, and the HLL code associated with each path of this graph can be executed immediately in response to a selection by means of simple mouse clicks.

MetaFrame is a *workbench* for interactive experimentation with tool coordination. High-level on–line HLL programming with synthesis support for complex linear portions (taking, in particular, the interface problem off the user) allows people with different programming skills and professional profiles to profitably develop and test even complex tool coordination structures in a comfortable, intuitive manner. This feature is complemented by specific support for visualization, through the graphical user interface providing comfortable tool data presentation through graph windows, text boxes, compact menus, but also for documentation and navigation in the repository of available tools, through an advanced hypertext system which provides on–line browsing of HLL code, as well as of tool or MetaFrame documentation (see [25]).

3 Tool Integration

The ease of coordination capitalizes on the capability of accessing heterogeneous tools in a uniform, abstract way as function calls in HLL programs which, via their arguments, pass abstract data to the underlying tools. Tool integration provides a means for adapting the reality of a heterogenous, therefore complex treasure of tools and components to this abstract view and mechanism, by masking the differences through tool-specific *adapters* programmed according to a strictly disciplined methodology.

In principle each pair of potentially cooperating tools requires both conversion of *data* and management of *control*. In contrast to previous approaches which either invested a lot of programming for coupling single tools through specialized converters (which obviously does not scale up) or a lot of effort in standardization of e.g. data formats (which was not very successful in the past because of the impossibility to obtain agreement between different communities), we decided to leave toolmakers their freedom by defining an abstract, coordination layer at which the coordination takes place and which is fairly independent of the concrete formats, mechanisms, paradigms, programming languages the actual tools adopt. This way people are totally free to choose their own formats and interfacing mechanisms, while we maintain a large potential for reuse on the coordination specific layer.

Technically, the practicality of this *integration policy* hinges on the appropriate use of *tool adapters* and the support of the *adapter generator*, which constitutes the current automatic integration support of our system.

The Tool Adapter. Each tool is integrated via an individual *tool adapter*, mapping the tool's own data formats, interfacing mechanisms, and offered functionalities into entities accessible to the HLL–Interpreter. Given a new tool,

- for each tool data type, the tool adapter instantiates a generic *tool data wrapper* which is itself defined in the METAFrame source code. Only by implementing HLL data values through uniformly wrapped tool data, easy integration of very different kinds of tool data is combined with HLL coordination that abstracts from all technical details. The adapters wrap tool data values into abstract data values before they pass them to the HLL–Interpreter for coordination, and they unwrap HLL values when they are passed to tools as tool input data.

- for each tool functionality, the adapter implements a C++ function that maps calls to the HLL function representing the tool functionality into the corresponding tool invocations, including the conversion of the HLL function arguments into tool input data and of tool output data back into HLL function return values.

When an adapter is loaded into METAFrame, it registers new HLL functions that represent the different invocable tool functionalities as well as newly defined HLL data types corresponding to the tool's concrete types. Registration means that the adapter calls the HLL–Interpreter to make the types and functions known. Note that the combination of separate compilation and dynamic linking allows users to integrate new tools into the running system without requiring a restart or even a recompilation of METAFrame.

Because of the delegation of the tool–specific parts into the tool adapters, the integration mechanisms can be tailored individually to every tool's needs, without requiring access to the tool's internals. The tool does not need to be modified in order to be integrated.

The Adapter Generator. To support the integration of new tools, an *adapter generator* (see Figure 2) may be used to generate the C++ adapter code from high–level adapter specifications instead of explicitely programming it in C++. Adapter specifications abstract from many technical details of adapter implementation.

Following the same trade-off principle already ilustrated between coordination and integration, the effort invested once into the generator pays off repeatedly in the (relatively more frequent) process of adapter writing. However, the technical knowledge about control and data passing as well as data conversion is specific to the single tools, so that it cannot be built-in into the adapter generator. Even in the high–level adapter specifications, these conversions have to be coded directly in C++.

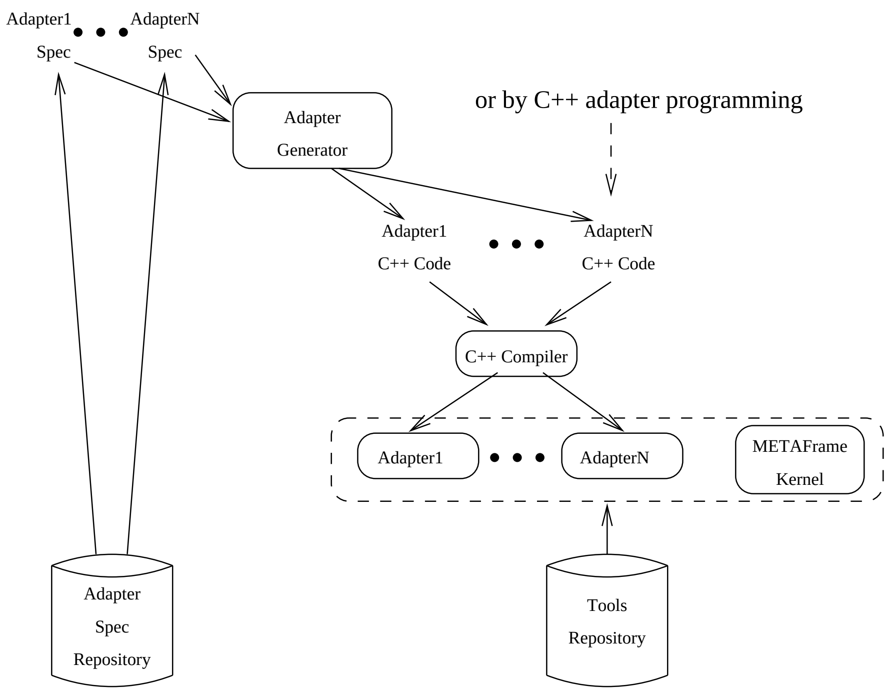

Figure 2: The Tool Integration Phase

4 The Coordination Environment

The global architecture of the MetaFrame environment is reported in Figure 3. The core of the coordination aspect is located in the Tool Coordination System, and it comprises the coordination language HLL and its interpreter (discussed in Sect. 4.3). The Tool Coordination System interacts with the Graphical User Interface, the Synthesis Tool and the Hypertext, as described in Sect. 4.1). In addition to graphical visualization facilities, the GUI (itself based on primitives contained in the GUI Library) also offers specific editors for the graphical entry of higher-level coordination specifications.

At the tool integration level, MetaFrame open architecture offers a (growing) number of application-specific tool libraries. In particular, the whole GUI is implemented as a distint application library, and also the Synthesis Tool and the translators between the coordination specification languages are part of the Computer-Aided Tool Synthesis Library (CATS). Other currently active applications include a collection of Dataflow and optimization algorithms for compiler construction, a synthesis-supported Requirement Engineering environment, a Service Definition environment for Intelligent Network Services (now marketed as Siemens' INXpress Advanced Service Definition environment), and the library of integreted tools of the Springer STTT ETI platform.

Each library distinguishes the actual tools, stored in an application-specific tool repository, and the coordination and synthesis-oriented tool meta-data repository, including e.g.

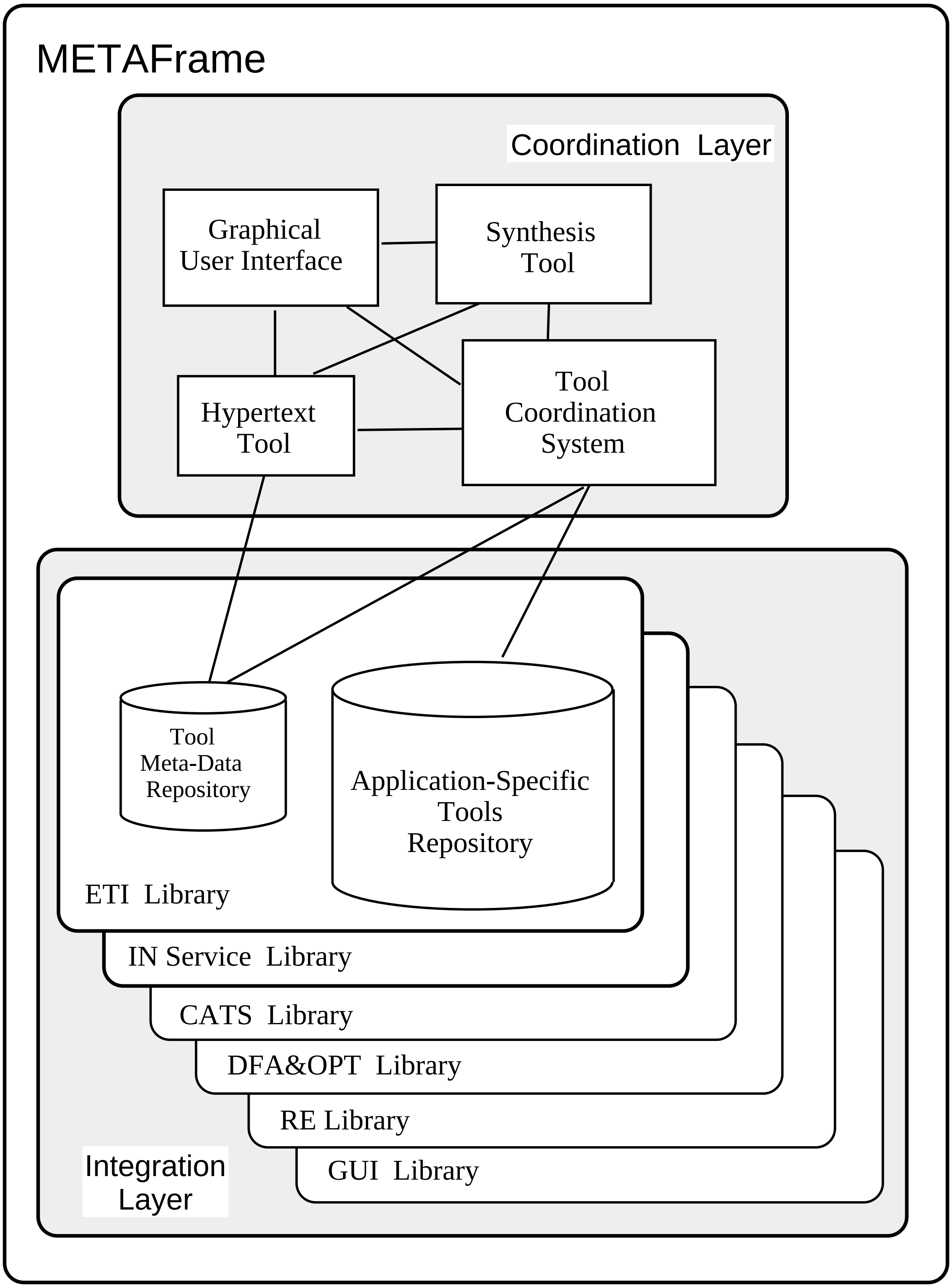

Figure 3: The METAFrame System Architecture

the tool adapters (adapter specification, C++ code, and compiled object code), the tool data wrappers, and the taxonomic information needed for the synthesis and for handling qualified types (see sect. 5.1).

4.1 High-Level Specifications and Synthesis Support

Users can directly experiment with tool coordination through on–line HLL programming in the shell window. However, the METAFrame graphical user interface, implemented using Tcl/Tk [20] and the FFGraph graphic library [12], features comfortable mechanisms for data presentation, documentation, and retrieval.

In addition to providing on–line browsing of documentation, the METAFrame hypertext system plays a central role in the coordination specification method based on automatic synthesis of tool invocation sequences from loose, abstract modal-logic specifications (see [26]). The modal-logic specification language and the synthesis algorithm in fact refer to abstract descriptions of the tools' functionality and of their interface, which are organized as taxonomies and accessible through the hypertext.

As an example, in [26] we showed how to use automatic synthesis in order to generate a pre-reducing preorder checker for pairs of CCS programs on the basis of 1) an efficient model-checker for mu-calculus formulas and labelled transition systems and 2) a library of transformation and minimization tools organized in a taxonomic way.

Graph windows are extensively used in the editors for Message Sequence Chart and Call–Flow coordination languages. Since coordination specifications entered in these two formalisms are translatable into the kind of modal-logic formulas the synthesis algorithm handles, these high-level graphical input languages offer easier to master alternatives to the modal-logic specification but still full automatic synthesis support.

The solution space generated by the automatic synthesis tool is displayed as a graph whose paths represent alternative tool invocation sequences satisfying the modal-logic formula (or the corresponding Message Sequence Chart or Call–Flow). Automatic layout algorithms and inspectors for each node (tool) and edge (interface description) in the solution graph ease the visualization of complex tool combinations and the presentation of the data associated with the graphical objects.

Each synthesized path corresponds to an executable HLL coordination program, which can be executed directly from the solution graph as soon as the selected path is unambiguous. This way, METAFrame realizes a coordination environment in which no traditional programming and no knowledge of the HLL and of the tools are needed: from the graphical specification of a coordination task to its execution and compilation, users are able to work without any knowledge of METAFrame's coordination mechanisms, language or interna.

Even a certain degree of interaction is immediately possible, since text windows, file selector and input boxes can be invoked from HLL code for access to and input of text and data during an execution run.

```
   (* Parse the input strings using the CCS and Modal tools ...    *)
var CCS_Term: t := CCS.parse(Win.inputBox("Please input CCS Term"));   (1)
var Prop: phi := Modal.parse(Win.inputBox(                             (2)
                                     "Please input Logic Formula"));
var Lts: ts, v; (* ... labeled transition systems                 *)
var Bool: success;
   (* Invoke the transition system generator tool LTS_Gen ...      *)
ts := LTS_Gen.generate_transition_system(t);                           (3)
   (* Invoke the ModelChecker tool ...                             *)
success := ModelChecker.model_check(ts, phi);                          (4)
if (not success) then
   (* Invoke the Abstraction tool to generate an error view ...    *)
   v := Abstraction.generate_view(phi, ts);                            (5)
   (* Use the METAFrame GUI to display the transition system ...   *)
   Graph.display(v);                                                   (6)
fi;
```

Figure 4: Example of a Coordinating HLL Program

Example As a concrete example of synthesized coordination, Figure 4 shows a slightly modified version of the synthesis example of [26]: a HLL program which coordinates the invocations to

(1) a parser for CCS expressions *CCS.parse*,

(2) a parser for modal-logic formulas *Modal.parse*,

(3) a generator of labelled transition systems from CCS terms on the basis of the operational semantics *LTS_Gen* in order to allow the use of

(4) a model checker for transition system models *ModelChecker* and, conditionally in case of error detection, the automatic generation of an error view for diagnosis purposes by means of

(5) the *Abstraction* package, which offers property-driven model collapse as one of its functionalities, and

(6) the METAFrame GUI through the *Graph* module, providing graph windows for comfortable presentation of graphs.

Here the desired CCS term and modal-logic formulas can be input interactively during execution in appropriate pop-up windows.
Depending on the level of integration, tools may use MetaFrame's GUI features, or also (additionally) their own GUIs, if they provide one.

Since the core of tool coordination lies at the level of HLL programs, central components are the HLL coordination language, its interpreter and the tool adapters, all implemented in C++, which are described in the next sections.

4.2 The Coordination Language HLL

The tool coordination language in MetaFrame is the **H**igh–**L**evel **L**anguage (HLL), an imperative *meta–level* language explicitely designed for coordination purposes: Abstracting from all technical details of the tool invocation mechanisms and intertool data-passing, it guarantees a uniform, meta–level coordination view independently of the particular implementation of the participating tools.

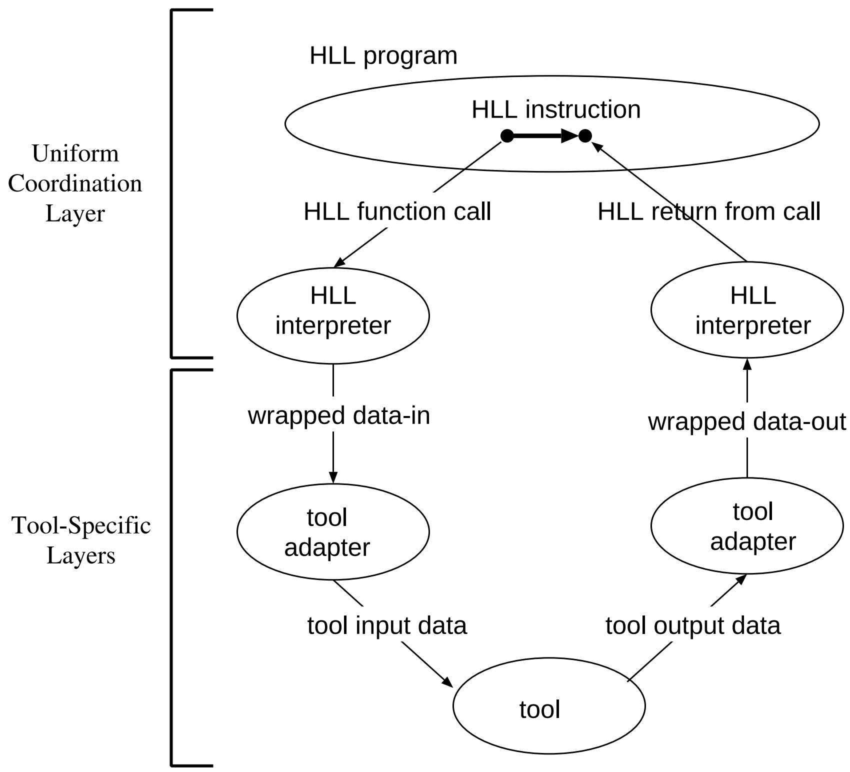

Figure 5: Execution of HLL Function Calls

Its aims differ from the typical imperative programming purposes since no computation should be implemented at the HLL level; rather all computation is to take place in the tools, so that the coordination and the computation aspects are strictly decoupled. In HLL, modules[3] represent *tools*, function calls represent *tool invocations*, while HLL data values represent *tool data*. Intertool communication is realized in form of HLL function calls with parameter passing, as sketched in Figure 5: the argument values in a function call are unwrapped and passed to the called tool as input data through the tool's adapter, which also transforms the returned data values back into the HLL function's wrapped return values.

The HLL module system as well as its control structures (like `if-then-else-fi` and `while`) allow an intuitive specification of complex coordination tasks, while abstract data structures (like tuples, sets and lists) and HLL variables allow comfortable management of tool data.

As already mentioned, the imperative, sequential design of the HLL complies well with METAFrame's automatic synthesis mechanism. On–line HLL programming in the shell window allows animation, interactive experimentation with and testing of HLL code.

Since METAFrame's GUI features (like graph windows or text boxes) are accessible through HLL functions (see Appendix A), the coordination environment provides graphical interaction capabilities through on–line HLL programming of the GUI comfortable presentation during interactive coordination experiments. Moreover, interactive GUI primitives like input boxes can also be used as HLL–level *user menus* inside the coordinating program, so that the control–flow of the coordination task may depend on run-time user input.

Since the coordination power of HLL programs is determined by the module and high–level data structure libraries, it grows together with the number of integrated tools: through its adapter, each tool contributes to the HLL its own invocation functions as well as (if not yet available) appropriate abstract data types representing its interface. HLL–level type checking guarantees safety and compatibility of intertool data-passing in this fluid and dynamic coordination setting.

4.3 The HLL–Interpreter

The HLL–Interpreter differs from conventional interpreters for imperative programming languages in the implementation of the mechanisms dealing with HLL data types, data values and functions: In these respects in fact it interacts closely with the tool adapters.

Accordingly, the HLL–Interpreter offers mechanisms to allow the adapters to register HLL functions and data types upon METAFrame startup. The HLL–Interpreter implements a *generic C++ wrapper for tool data*: the generic wrapper class is an abstract base class defining the uniform interface that all wrapped data values are required to offer to the coordination layer so that they can be used in HLL programs. For every tool data type, the adapter instantiates the generic wrapper class through C++ object–oriented inheritance. The specific instance adds the specific wrapped tool data to the generic wrapper. Since every HLL data value implementation `is-a` wrapper, at the interpreter

[3] The HLL has a module system: all function calls are qualified by the module name.

level wrappers are the implementation of HLL data values and are treated *polymorphically*. Given that at the coordination level there is no computation, but only intertool communication, this point of view is sufficient: the interpreter only needs storing wrapped tool data in HLL variables and passing them to other tools as HLL function arguments and return values. This way, concrete tool data are not interpreted at the HLL level, and the implementation of all the mechanisms that require interpretation of the wrapped tool data as well as wrapping and unwrapping of tool data is delegated to the adapters (see Figure 5).

5 Flexible Data Integration

The METAFrame approach allows choosing between different degrees of abstraction during tool data integration, to deal with the fact that

- tools **implement** data types differently, so that (syntactic) conversion is necessary when data is passed between tools, and
- tools **interpret** data differently, e.g. at the semantic level, the same graph may be interpreted by one tool as a program control flow graph, while another one may interpret the same data as term graph of some process algebra.

To cope with this de facto situation, METAFrame supports a variety of (data) integration approaches, which may be broadly grouped according to how deep into the coordination mechanism the differences due to different data representation/interpretation conventions penetrate.

Deep Tool Data Integration. Here *each* tool's implementation and/or interpretation of an interface data type is wrapped and registered as a separate HLL data type. As a consequence, during coordination on the HLL level, data type conversions must be invoked explicitely if data is to be passed between tools. Here the coordinator may profit of the synthesis-based automatic type conversion offered by METAFrame: the coordinator does not need to have detailed knowledge about the conversions, rather one may loosely specify his needs, e.g. through a modal-logic formula or graphically, as Message Sequence Chart or as Call Flow, and let the synthesis algorithm construct the needed sequence of conversions and automatically generate the HLL program for their invocations (see Sect. 4.1 and Figure 3).

Shallow Tool Data Integration. This approach is much more abstract, since it registers only a *single* HLL data type for each collection of (related) tool data types. The HLL type is then abstractly implemented through a METAFrame–internal data representation, which holds all the information of the represented tool data types. The conversion between different tool data types is dealt with at the adapter level: the adapter converts tool output data into the METAFrame–internal format before passing it to the HLL–Interpreter, and vice versa. This way, HLL programmers and coordination users are not encumbered with explicit conversion handling at the coordination level.

5.1 The Qualified Types

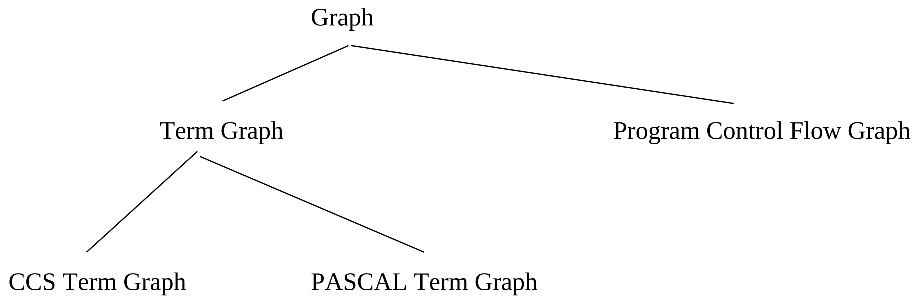

Figure 6: Example Taxonomy for Qualified Types

The shallow integration approach is particularly attractive for dealing with differing interpretations of data by the tools. The *qualified types* concept classifies the different interpretations in a taxonomy similar to those used in [26], with the HLL type being the root of the taxonomy. Figure 6 reports the qualified type taxonomy for the *graph* example already mentioned. Every value of the HLL type is internally qualified with its detailed interpretation. The determination and interpretation of the qualified types is located in the adapters, as they know how the output of their tool may/must be interpreted. The adapters are here the type controlling instance: they may e.g. export the most liberal typing and refuse importing incorrect typings.

The central advantage of this concept is its combination of flexibility and control. The taxonomy of qualified types is accessible during tool coordination through the hypertext system, so that the coordinator may interactively "collapse" the type taxonomy, restricting the degree in internal qualification and thereby the differences in data interpretation. This directly affects the *compatibility* of tools: fewer qualifications mean that more and more tools become compatible, making new coordination structures possible. It also affects the solution space of the synthesis tool: on the one hand higher tool compatibility means that result graphs may become larger, since more combinations are allowed, but on the other hand less explicit conversions may be necessary, since some may be automatically taken care of in the adapters.

The qualified types concept allows the user to dynamically alter type compatibility and thereby restrict tool coordination according to his *experience* with the tools. While a user with very little experience may seek a high degree of type qualification, so that only "safe" combinations of tools — where the tools agree on the interpretation of data — are possible and the synthesis will give a small set of alternative solutions, an experienced user may restrict qualification, allowing him to experiment with unconventional tool coordinations.

Our experience shows that the combination of tools from heterogeneous application areas can be very successful, e.g. in [28] and [16], model checking — a method for verifying distributed systems — was used in the service design for Intelligent Telephone Network Services and in Data Flow Analysis with great success.

6 Applications and Experience

The strength of METAFrame's tool coordination philosophy is being able to integrate heterogeneous tools, implemented in a number of languages and of programming paradigms (functional, imperative, object-oriented) and to let them communicate using different mechanisms too. The concept of tool adapters is central, since it allows the METAFrame integration mechanisms to be tailored individually to each tool's needs.

A variety of tool integrations was already applied successfully in practice, spanning the whole spectrum between deep and shallow integration, and manual to synthesis-supported coordination.

- Tools like the *Fixpoint Analysis Machine* model checker [22] or the automatic *Synthesis Tool* (used for coordination support itself), being implemented in C, C++ or even PASCAL or Fortran, can be incorporated directly into the METAFrame executable via dynamic linking. The advantage of such tight integration is that the control and data passing mechanism between a tool and METAFrame (though through the tool adapter) can be directly a C++ function or method call. This guarantees speed, and is therefore used for such time–critical tools as the Model Checker.

- Tools implemented in different interpreted programming languages are integrated by either 1) treating the combination of interpreter and tool code as a separate executable that is invoked to process the tool code, or if the interpreter is embeddable, it is 2) linked into the METAFrame executable and accessed via some `interpret()` function from its application programming interface (API). In both cases, data conversion is necessary when data is passed to and from the tool.

  The former concept was successfully applied during tests with a new METAFrame *GUI* written in Java [29]. The Java interpreter was executed as a separate tool, interpreting the tool code written in the Java language. Passing of data between the adapter and the Java interpreter was performed using `UNIX` sockets. Successful tests include several sessions allowing users to remotely work with METAFrame from the North Carolina State University in Durham (USA).

  The METAFrame *hypertext system* constitutes a successful application of the second schema, with the interpreted language being Tcl and its interpreter being integrated by linking it into the METAFrame executable.

- Tools implemented as separate executables are loosely integrated. They are either invoked by their adapter using shell calls (e.g. `system()` calls in `UNIX`) or may be running continuously, as a server waiting for requests, and are only connected to by the adapter, playing the role of a client. This form even allows METAFrame and the tools to be running on different machines with remote communication over the network. Data communication in both directions can be performed in a number of ways: using files, operating system services like sockets in UNIX or interoperability services like CORBA or Java RMI.

A number of successful applications of METAFrame have already made use of this concept:

- Both forms of integration were successfully tested on the RAP term rewriting engine [13]. In one version, RAP is restarted for every request with data (here entire programs) being communicated through files, in another version RAP runs continuously as a server with data communication through sockets.
- In the CATS–METAFrame application [26, 23], the LOLA deductive database server was integrated using socket communication. Tests as well as on-line demonstrations were successfully carried out with METAFrame running in Aarhus (DK) and Saarbrücken (D), LOLA running in Passau, and communication happening over the Internet.
- In a newer version of the Java implementation of the METAFrame GUI, Java Remote Method Invocation (RMI) is used as the interoperability service between the adapter and the Java interpreter.

Tool coordination methods and their relationship with different integration methods are the central topic of the ETI project. Here we intend to gather experience about tool coordination and develop standard mechanisms for coordination-oriented tool integration.

7 Discussion

To our knowledge, METAFrame is the only coordination environment offering the following facilities:

- a coordination language, the HLL, tailored for programming even complex coordination tasks at an abstract, i.e. implementation-independent level, together with the corresponding interpreter
- high-level languages for coordination specifications, including modal logic as well as visual formalisms widely used in the industrial practice,
- automatic synthesis of (sequential) coordination programs from loose specifications given in either of these formalisms
- flexible mechanisms for the ad-hoc integration of tools, based on the fine-grained abstraction concepts implemented through tool adapters, tool data wrappers, and qualified data types, which altogether grants

- open system coordination, allowing the dynamic, even on-line expansion and modification of the repository of tools without requiring recompilation.

This profile qualifies METAFrame as service definition environment also for heterogeneous, Internet-based services, since it offers all the functionalities mentioned in [11] and [7].

Related Work.

ToolBus In the TOOLBUS [5, 17], tools are integrated through adapters and coordinated through so–called **T** scripts in a process–oriented way: the coordination language is a process algebra with operators describing sequential tool invocations as well as parallel composition with synchronous or asynchronous (broadcasting) intrertool data communication.

Requiring expertise in expressing coordination through process algebra scripts, the TOOLBUS coordination language is less intuitive for the broad user, especially when it comes to on–line coordination. In [5] it is stated that larger practical applications made it necessary to extend the process algebra with programming language–like constructs as assignments, local variables and `if-then-else-fi`, thus coming closer to the imperative programming style behind the HLL. Moreover, with tool data being represented through terms, there is no typing in **T** scripts to check data compatibility, and while parallel composition allows elegant coordination, there is no automatic support mechanism like synthesis to support loose coordination specifications and to overcome tool data incompatibilities at the coordination level, e.g. by automatic generation of tool invocation sequences that perform tool data transformations. The TOOLBUS also lacks visualization mechanisms like hypertext or elaborate GUI, that actively support the user in the interactive tool coordination and allow comfortable presentation of tool data.

Linda Linda [10], one of the first coordination languages, is based on a blackboard principle, implemented by means of shared tuple spaces for data communication and tool synchronization. Being a coordination language only, it is more adequately compared with the HLL than with the whole METAFrame. Recent developments along its lines include the object-oriented OIL [15] language, which introduces an abstract layer for data communication, thus allowing open system coordination, which is also the aim of LAURA [30].

CORBA, (D)COM, RPC, Java RMI Interoperability services like CORBA [19] or (D)COM [18] allow software components to exchange data and invoke other component's functionalities over a network. The components may be implemented in different programming languages, and although object–orientation is the underlying principle of these services, the components may also be implemented in imperative languages like C. Java Remote Method Invocation (RMI) offers the same set of services, but restricted to the Java programming language.

While these services implement elaborate mechanisms for data and control integration, they do not support any flexible coordination. Tools are themselves in charge of the coordination, by taking care of the initiation and correct exchange of control and data. This mechanism resembles a net of mutual procedure calls, with the interoperability service being used as the network transport mechanism for control and data. Neither flexible reconfiguration of the coordination structure nor interactive development of coordination in a workbench environment are possible: There is in fact no notion of a meta-level coordination instance or programming, and there is no high–level global control over the flow of control and data.

Remote procedure calls (RPCs) work along a similar principle, the difference being that the RPC mechanism is less elaborate (nested RPCs are a problem) and lacks an object–oriented approach.

Broadcasting and Message Passing Environments In broadcasting–based environments like Field [21], Sophtalk [6] or SPLICE [8], tools are independent agents that interact using broadcast communication. In ToolTalk [31], Polygen [33] and Manifold [1], intertool communication is based on message passing. In both approaches there is no global support for tool coordination as in the METAFrame workbench: they lack a central instance that actively coordinates the tools according to a global description, specifying the global coordination structure and possibly even allowing on–line reconfiguration of the coordination structure. Rather, single tools react to incoming messages which have to be issued ad hoc by the users.

Software Development Environments Software Development Environments (SDEs) like MARVEL [3], OZ [32] or SPADE [2] coordinate compilers, linkers or debuggers as external tools. In these environments, tool coordination focuses on the software development process and the tools come from that application area only.

Wrapping is used to connect the tools to the SDE, but the connection between the wrapper and the SDE is loose and there is no passing of complex data between both sides: while MARVEL and OZ use shell scripts as wrappers and exchange success/failure status values and file names between the wrapper and the environment, SPADE uses the DEC FUSE environment with its signaling mechanism. Tool data are kept externally in files or databases, the SDE only deals with file names or database references as symbolic representations and does not handle the data internally.

The coordination languages are Petri net-based (SPADE) or rule-based (MARVEL, OZ). Because tool data is kept externally only, there is no notion of data management or data types in these languages. In the case of the SPADE environment, the coordination language concentrates on the cooperation of the human software developers (Computer Supported Cooperative Work CSCW) and only indirectly addresses the underlying tools.

Coordination in SDEs focuses on *homogeneous* tools and data from the area of software development. METAFrame however aims at coordinating *heterogeneous* tools and data. Through the automated synthesis mechanism and the qualified types approach, METAFrame explicitely addresses complex data conversion issues.

Wrappers and Adapters in Databases Database architectures like Garlic [9] use adapters (called "wrappers" there) to connect database tools to a central query engine. These architectures are data-oriented and focus on providing high-level query services for heterogeneous data. The "coordination language" is a high-level query language, there is no notion of control or combination of tool functionality in the language. When the query engine executes queries, it submits a fixed set of retrieval requests to the adapters which translate the requests into the database-specific retrieval implementation.

These database architectures can be seen as instances of the MetaFrame environment where the tools are databases, the set of tool functionalities is restricted to some fixed retrieval operations, and the coordination language specifies database queries only. While MetaFrame deals with complex control structures in coordination, the database architectures focus on data only.

References

[1] F. Arbab, I. Herman, P. Spilling: "*An overview of Manifold and its implementation*", Concurrency: Practice and Experience, Vol. 5, pp. 23-70, 1993.

[2] S. Bandinelli, E. Di Nitto, A. Fugetta: "*Supporting Cooperation in the SPADE-1 Environment*", IEEE Transactions on Software Engineering, 22(12), pp. 841–865, 1996.

[3] Naser S. Barghouti: "*Supporting cooperation in the* Marvel *process-centered SDE*", Proceedings of the Fifth ACM SIGSOFT/SIGPLAN Symposium on Software Development Environments, ACM SIGSOFT Software Engineering Notes, 17(5), pp. 21–31, 1992.

[4] M. von der Beeck, B. Steffen, T. Margaria: "*A Formal Requirements Engineering Method and an Environment for Specification, Synthesis, and Verification*", Proc. of SEE '97, 8th IEEE Conference on Software Engineering Environments, Cottbus (Germany) 8-9 April 1997.

[5] J. A. Bergstra, P. Klint: "*The ToolBus Coordination Architecture*", Proc. 1st Int. Conf. on Coordination Models and Languages (COORDINATION'96), LNCS 1061, pp. 75-88, Springer Verlag, 1996.

[6] J. Bertot, I. Jacobs: "*Sophtalk tutorials*", Technical Report 149, INRIA, 1993.

[7] M. Bieber, F. Vitali: "*Toward Support for Hypermedia on the World Wide Web*", IEEE Computer, January 1997, pp. 62-70.

[8] M. Boasson: "*Control systems software*", IEEE Transactions on Automatic Control, 38(7), pp. 1094-1106, 1993.

[9] M. J. Carey, L. M. Haas et al: "*Towards Heterogeneous Multimedia Information Systems: The Garlic Approach*", Research Issues in Data Engineering, IEEE Computer Society Press, pp. 124–131, March 1995.

[10] N. Carriero, D. Gelernter: "*Linda in context*", CACM, 32(4), pp. 444-458, 1989.

[11] P. Ciancarini, R. Tolksdorf, F. Vitali: "*Weaving the Web Using Coordination*", Proc. Coordination'96, Cesena (I), April 1996, LNCS 1061, Springer, pp. 411-415.

[12] C. Friedrich: "*The* ffgraph *Library*", Techn. Rep. MIP95-20, Univ. of Passau (D), December 1995, http://www.fmi.uni-passau.de/ friedric/ffgraph/main.shtml

[13] A. Geser, H. Hußmann: "*Experiences with the RAP system — a specification interpreter combining term rewriting and resolution*", Proc. 2nd ESOP, LNCS 213, pp. 339-350, Springer Verlag, March 1986.

[14] ITU-T Recommendation Z.120: "*Message Sequence Chart (MSC)*", March 1993.

[15] T. Kielmann: "*Designing a Coordiantion Model for Open Systems,*" Proc. Coordination'96, Cesena (I), April 1996, LNCS 1061, Springer, pp. 267-283.

[16] M. Klein, J. Knoop, D. Koschützki, B. Steffen: "*DFA&OPT*-METAFrame*: A Tool Kit for Program Analysis and Optimization*", Proc. TACAS'96, Passau (D), March '96, LNCS 1055, pp. 422-426, Springer Verlag.

[17] P. Klint, P. Olivier: "*The TOOLBUS Coordination Architecture: A Demonstration*", Proc. of 5th Int. Conference on Algebraic Methodology and Software Technology (AMAST'96), LNCS 1101, pp. 575-578, Springer Verlag, 1996.

[18] Microsoft: "*The Component Object Model Specification*", http://www.microsoft.com/oledev/olecom/title.htm.

[19] OMG: "*The Common Object Request Broker: Architecture and Specification Revision 2.0*", Technical Document ptc/96-08-04.

[20] John K. Ousterhout: "*Tcl and the Tk Toolkit*", Addison–Wesley, April 1994.

[21] S. P. Reiss: "*Connecting tools using message passing in the Field programming environment*", IEEE Software, 7(4), July 1990.

[22] B. Steffen, A. Claßen, M. Klein, J. Knoop. T. Margaria: "*The Fixpoint Analysis Machine*", (*invited paper*) to CONCUR'95, Pittsburgh (USA), August 1995, LNCS 962, Springer Verlag.

[23] B. Steffen, B. Freitag, A. Claßen, T. Margaria, U. Zukowski: "*Intelligent Software synthesis in the* DaCapo *Environment*", Proc. 6th Nordic Workshop on Programming Theory, Aarhus (Denmark), BRICS Report N. 94/6, December 1994.

[24] B. Steffen, T. Margaria, V. Braun, N. Kalt: "*Hierarchical Service Definition*", appears in Annual Review of Communication, Int. Engineering Consortium Chicago (USA), IEC, 1997.

[25] B. Steffen, T. Margaria, A. Claßen, V. Braun, V. Kriete, M. Reitenspieß: "*A Constraint-Oriented Service Creation Environment*", Proc. of TACAS'96, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Passau (Germany), LNCS 1055, pp. 418-121, Springer Verlag, March 1996.

[26] B. Steffen, T. Margaria, A. Claßen: *Heterogeneous Analysis and Verification for Distributed Systems*, "SOFTWARE: Concepts and Tools" Vol. 17, N.1, pp. 13-25, Springer Verlag, March 1996.

[27] B. Steffen, T. Margaria, A. Claßen, V. Braun: "*Incremental Formalization: a Key to Industrial Success*", in "SOFTWARE: Concepts and Tools", Vol.17, N.2, pp. 78-91, Springer V., July 1996.

[28] B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß: "*An Environment for the Creation of Intelligent Network Services*", invited contribution to the book "The Advanced Intelligent Network: A Comprehensive Report", Int. Engineering Consortium Chicago (USA), Dec. 1995, pp. 287-300. – also reprinted in the *Annual Review of Communications*, IEC, 1996.

[29] SUN Microsystems: "*Java(tm): The Java Language Specification*", 1995.

[30] R. Tolksdorf: "*Coordinating Services in Open Distributed Systems with* LAURA", Proc. Coordination'96, Cesena (I), April 1996, LNCS 1061, Springer, pp. 386-402.

[31] "*Designing and writing a ToolTalk procedural protocol*", Technical Report, SunSoft, June 1992.

[32] G. Valetto, G. E. Kaiser: "*Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments*", Proc. 7th Int. Workshop on Computer-Aided Software Engineering, Toronto (Canada), IEEE Computer Society Press, pp. 40-48, July 1995.

[33] E. L. White, J. M. Purtilo: "*Integrating the heterogeneous control properties of software modules*", Proc. 5th ACM SIGSOFT Symposium on Software Development Environments, pp. 99-108, Software Engineering Notes, Vol. 17, 1992.

A A Coordination Example

In this section we demonstrate METAFrame's integration and coordination methodology on a simple example involving two tools taken from our experience in DFA&OPT–METAFrame, the Dataflow Analysis application described in [16]. In particular, we intend to show how an application-specific tool like the *CFG_Generator* can interact with MetaFrame's graphical primitives, which are all available in a distinct tool library.

- The *CFG_Generator* generates program control flow graphs from source code in a PASCAL–like programming language. The tool is available in form of a separate executable program that takes its (textual) input, a concrete program, from a file and outputs the control flow graph to another file.
- The MetaFrame*Graphical User Interface* features comfortable mechanisms for data presentation, documentation, and retrieval and in particular it offers primitives for displaying graphs in graph windows.

The following subsections give detailed descriptions of the tool adapter specifications for the two tools, followed by their implementation and by the description of a simple coordination example.

A.1 The GUI Tool Adapter

GUI functionality has to be made available to the coordination environment by means of an appropriate adapter. Since the GUI tool is part of the MetaFrame executable, the functionality is directly accessible through a `C++` method call in the body implementing the procedure.

Here we show how writing the concrete adapter can be supported by the automatic adapter generation facility. To this aim, a concise specification is transformed by the adapter generator into the `C++` adapter code.

```
%header { #include "FFGraph.h" }                                        (1)
%type FGraph { FFGraph value; }                                         (2)
%eq FGraph (g1, g2) { RETURN = (g1.value == g2.value); }                (3)
%procedure display (ref FGraph: g) { g.value.display(); }               (4)
```

Figure 7: The Adapter Specification for the `GUI` Tool

Specifying the Adapter. Adapter specifications are written in a comfortable, higher-level language similar to that in parser specification files for the `yacc` parser generator. Figure 7 reports the adapter specification[4] for the graph display function offered by the MetaFrame GUI. Here,

[4]In this and in the following code fragments the line numbers on the right–hand side are added for explanation purposes.

(1) The `%header` section contains `C++` code that will be included (uninterpreted by the generator) into the generated `C++` adapter code. It is used to include `C++` header files or define auxiliary `C++` functions needed by the `C++` code in the bodies of `%procedure` or `%function` definitions. The `%header` mechanism is similar to the one in parser specification files for the `yacc` parser generator which is used there for the same purposes.

(2) The `%type` line defines a new HLL data type `FGraph` representing a tool data type. It states that `FGraph`s are implemented as wrapped `C++` `FFGraph`s. The class `FFGraph` is implemented in the `C++` code of the GUI tool.

(3) The `%eq` line defines a HLL equality test on `FGraph`s. Such a test is needed by the implementation of HLL `set`s, therefore an `%eq` operation must be defined for every HLL type defined through `%type`. For an explanation of the `RETURN` in the body of the `%eq` definition see the next item.

(4) The `%procedure` section defines a new HLL procedure `display()` that represents the GUI tool's `display()` functionality. The HLL procedure takes a HLL `FGraph` reference argument. The body of the `%procedure` section consists of `C++` code that will be included into the adapter code implementing the HLL function.

The GUI tool is part of the METAFrame executable, therefore the functionality is directly accessible through a `C++` method call in the body implementing the procedure (see next section for an example of a more complicated tool functionality invocation).

The generator allows to specify the code on a more comfortable, higher level: The argument variables are treated specially by the generator. In the example, `g` represents a HLL–level variable, while `g.value` represents the wrapped `C++` tool data. Code for unwrapping `g` into `g.value` is automatically inserted by the generator into the generated output. A similar mechanism is provided for the return value of HLL functions through the "magic variable" `RETURN`, where code wrapping the tool data value into the HLL return data value is automatically generated.

That mechanism is similar to that in parser specification files for the `yacc` parser generator where the nonterminals are "implemented" through C code containing `$` variables that are automatically dealt with by the parser generator.

A.2 The CFG_Generator Tool Adapter

The `CFG_Generator` tool contained in the DFA library has its own format for storing output control flow graphs in files. Consequently, there are two design options in designing the adapter:

- either the private format is registered on the HLL level as a separate tool data type, leading to a *deep integration* of the data type,

- or we use some already defined HLL graph type, in this case the `FGraph` type, thus leading to a *shallow integration*.

The second option was chosen in practice, so that the adapter must include the conversion from the `CFG_Generator` graph format into the `FFGraph` format, which is the implementation format of the HLL `FGraph`s. The conversion happens through parsing of the `CFG_Generator` output file, where the parser explicitely constructs the corresponding `FFGraph` which is then wrapped and passed back to the HLL–Interpreter as a HLL `FGraph` value.

```
(1)  %require String FGraph
(2)  %header
(3)  {
(4)  #include <stdlib.h>
(5)  #include "FFGraph.h"
(6)  void write_to_file(char*, char*) { /* implement here */ }
(7)  FFGraph parse_CFG_Gen_file(char*) { /* implement parser here */ }
(8)  }
(9)
(10) %function generate (String: prg_string) : FGraph
(11) {
(12)   write_to_file(prg_string.value, "/tmp/gen.in");
(13)   system("CFG_Generator -i /tmp/gen.in -o /tmp/gen.out");
(14)   RETURN.value = parse_CFG_Gen_file("/tmp/gen.out");
(15) }
```

Figure 8: The Adapter Specification for the `CFG_Generator` Tool

A line–by–line description of the adapter specification given in Figure 8:

(1) The `%require` command declares HLL types that are used as argument or return value types for HLL functions or procedures defined (via `%function` or `%procedure`) in this adapter specification. The generator will consult the adapter specifications defining those types in order to be able to generate the correct wrapping or unwrapping `C++` code for the values of these types.

(2-8) For an explanation of the `%header` section, see section A.1. Here, `stdlib.h` is included because it declares the `system()` function that will be used in the body of the `generate()` function, while `FFGraph.h` is needed for the return value of `parse_CFG_Gen_file()`. The auxiliary `C++` functions `write_to_file()` and `parse_CFG_Gen_file()` needed in the body of the `generate()` function are also implemented here.

(10) The `%function` section defines a new HLL function `generate()` that represents the CFG_Generator tool's functionality. The HLL function takes a HLL `String` argument and returns an `FGraph`.

(11) The CFG_Generator tool takes its input from a file. Therefore, the unwrapped HLL `String` is written into a file.

(12) The CFG_Generator tool is invoked. The tool is implemented as a separate executable that has to be started by the adapter. `UNIX` operating system functionality is used for that via the `system()` call. Via command–line options `-i` and `-o`, the executable is instructed to read its input from the file `/tmp/gen.in` and write its output to `/tmp/gen.out`.

(13) The output file contains a graph in the CFG_Generator's own graph format. The file contents are read in, parsed and converted into a `C++ FFGraph` by the `parse_CFG_Gen_file()` function. The `FFGraph` is wrapped and returned to the HLL level as an `FGraph`.

The body of the `%function` section illustrates that the adapter specifications still require the tool integrator user to explicitely code all technical details of tool invocation and data conversion. Technical knowledge about these aspects is required, the adapter generator cannot offer any support in this respect because the details are tool dependent and have to be handled on a case-by-case basis. However, the adapter specifications hide the details of wrapping and unwrapping of values as well as the details of the registration of the new HLL types and functions with the HLL–Interpreter, which means that they abstract from adapter–specific technical details.

A.3 A Tool Coordination Example

The short HLL program coordinating the two example tools reported in Figure 9 allows any program in the PASCAL–like language, given as HLL `String`, to be passed to the `CFG_Generator` tool, which extracts the program control flow graph, represented on the HLL level as an `FGraph`. The `GUI` tool is then invoked to visualize the corresponding graph.

```
require("CFG_Generator");
require("GUI");
var String: prg := "a:=1; b:=2; if (a<b) then max:=a else max:=b fi;"
var FGraph: g;
g := CFG_Generator.generate(prg);
GUI.display(g);
```

Figure 9: The Coordinating HLL Program

The execution proceeds as follows:

(1),(2) The HLL–Interpreter is instructed to load the HLL modules `CFG_Generator` and `GUI`[5]. The modules are implemented through the adapters for the `CFG_Generator` and `GUI` tools, so these adapters are dynamically loaded into the METAFrame executable.

(3) A HLL `String` holding the example input program for the `CFG_Generator` tool is defined. This shows how easy it is to enter coordination input or test data on the HLL level.

(4) A HLL `FGraph` is declared. It is implemented through a wrapped `C++ FFGraph`.

(5) The `generate()` functionality from the `CFG_Generator` HLL module is invoked through a HLL function call. The module is implemented by the adapter for the `CFG_Generator` tool. It returns a HLL `FGraph` containing the control flow graph.

Internally, the functionality is implemented using a `system()` shell call for control passing and files for data passing. On the HLL level, these technical details are completely hidden. The fact that the generator uses its own graph format for its output graphs, making data conversion necessary, is also hidden from the HLL level.

(6) The `display()` functionality from the `GUI` HLL module is invoked through a HLL procedure call.

[5] The name of the HLL module is given through the name of the file containing the adapter specification.