ORCAS: Towards a CHR-Based Model-Driven Framework of Reusable Reasoning Components

Jacques Robin[1] and Jairson Vitorino[2]

Universidade Federal de Pernambuco, Centro de Informática, Caixa Postal 7851 CDU Recife PE 50732-970, Brazil,
{robin.jacques,jairson@gmail.com},
WWW home page: http://www.cin.ufpe.br/~orcas

Abstract. We present the long term vision and current stage of the ORCAS project which goal is to develop an easily extensible framework of reusable fine-grained automated reasoning components. It innovates in software engineering by putting forward the first integration of model-driven, component-based, aspect-oriented and formal development techniques. It innovates in automated reasoning by proposing to build the most varied AI applications and inference engines (e.g., for deduction, abduction, inheritance, belief revision, constraint solving, optimization and induction) by assembling components, each one encapsulating a CHR^v knowledge base and reusing a CHR^v inference engine as a universally shared server component.

1 Introduction

Over the last two decades, Automated Reasoning (AR) has undergone an impressive modernization resulting in ever more rigorously defined concepts and scalable techniques [16]. Over the same period, embedded AR functionalities have become the key competitive edge for many practical, real-world applications such as business intelligence, resource optimization, product recommendation and customized configuration, robotics, simulations and games. Yet, AR techniques remain highly underused in such commercial software. Why? It is mainly because the cost of developing and incorporating AR functionalities into mainstream software applications remains prohibitively high. This in turn is primarily due to the old school software engineering approach with which AR software continues to be developed and deployed today. It overemphasizes performance scalability while neglecting practical usability and ignoring reusability. It also essentially still conceives the development process as a pure programming task, overlooking architectural design, platform independent problem modeling, testing, and lifecycle evolution through incremental requirement extensions. In the AR context, usability has many facets. It includes support to visually explain the reasoning at multiple abstraction levels and to automate systematic knowledge base testing. Current AR software rarely provides such support. Usability also includes the ability to be deployed in multiple contexts and to be straightforwardly embedded within applications based on mainstream IT standards. This is also rarely the case of current AR software for it tends to be based on exotic, non-standard Prolog extensions with very limited middleware. Difficult to use, current AR software is even harder to reuse, for it generally comes as one large monolithic piece of source and executable code with no accompanying visual abstract model to clarify its design. This prevents reusing part of its functionalities in a new context and explains why each new inference engine is essentially built from scratch at a very high cost. This is a terrible waste since conceptually, most advanced AR techniques are really compositions of several previous techniques [15]. For example, first-order deductive logic programming essentially composes propositional backward chaining with unification. By further composing it with inheritance, Bayesian probability calculus, abductive hypothesis construction, inductive hypothesis construction or domain-specific constraint solving, one respectively obtains object-oriented logic programming, Bayesian logic programming, abductive logic programming, inductive logic programming and constraint logic programming. If each minimal reasoning task was encapsulated in a corresponding reusable software component, a wide variety of reasoning services, each one customized to the specific AR needs of a given application, could then be assembled at low-cost.

In this paper, we present the long-term vision and first concrete results of ORCAS (Ontology-driven Reasoning Components for Agent-based Simulations), an ongoing project that aims to develop a framework of such reusable AR components. These components must be easy to use as stand-alone software, easy to assemble to provide a variety of more sophisticated AR services and easy to integrate in the most diverse applications. In order to reach this goal, ORCAS is based on ten design principles:

1. Define an AR component framework made of five aligned elements: a component engineering method, a component meta-model, a component CASE tool, a set of target component deployment platforms, and a component library[1].
2. For the component method, leverage, integrate and adapt to the specificities of AR cutting-edge software reuse techniques such as model-driven development [13], component-based development [2], built-in contract testing [9] and aspect-oriented development [4];
3. For the component meta-model, define a UML2 profile [6];
4. For the component library, encapsulate within each component in addition to the main reasoning concern, the orthogonal concerns of testing, reasoning explanation and GUI;
5. For the component library, realize the main concern as a sub-component, and among the orthogonal concerns realize as sub-components those loosely coupled with the main one, and as aspects those tightly coupled with it;
6. For the component library, use rule-based constraint programming in the language CHR^v [1] as a unifying basis for formal semantics and assembly;
7. For the component library, stratify the artifacts describing each component in four levels: the knowledge level in UML2, the formal level, the implementation level in AspectJ, Java, C# and/or C++ and the deployment level as .jar or .dll files;
8. For the formal level of the component library, define an object-oriented extension of CHR^v, called CHORD (Constraint Handling Object-oriented Rules with Disjunctive bodies), based on Frame Logic [12] for the semantics of inheritance and its interaction with deduction and abduction [18];
9. For the component CASE tool, define model transformations from UML2 to CHORD (knowledge to formal level) and from CHORD to AspectJ, Java, C# and/or C++ (formal level to implementation level) and integrate them as plug-ins in UML2 CASE tools and Eclipse;
10. For the component deployment platform, target the mainstream de-facto standards Eclipse and Visual Studio.

In the rest of this paper, we motivate and elaborate each these principles in turn. We then briefly discuss the current state of the project and draw a roadmap for the next steps. We conclude by summarizing the envisioned contribution of the framework.

2 Model-Driven Aspect-Oriented Engineering of AR Components

In order to achieve the highest degree of extensibility, ORCAS leverages the most modern software reuse techniques, namely Model-Driven Development (MDD) [13], Component-Based Development (CBD) [2], Built-In Contract Testing (BICT) [9] and Aspect-Oriented Development (AOD) [4]. We overview each one of them in turn, before explaining how they are integrated in ORCAS.

Model-Driven Development (MDD) Put forward by the Object Management Group (OMG), MDD pursues two related goals. The first is to minimize the cost of deploying the same functionalities on a variety of platforms. The second is to automate an ever growing part of the development process. To achieve these goals, MDD proposes to raise the level of abstraction where most of the development

[1] While "component framework" has been used in previous research to denote any of these five elements alone, we use it here to denote their combination in a coherent whole.

effort is spent from source code to models, meta-models and model transformations. In MDD, development starts by constructing a Platform Independent Model (PIM) of the application. This PIM must fully refine the software design down to the level of instructions available as built-in by the most widely used platforms for a particular domain. In MDD, since refinement has entirely moved up to the PIM level, implementation thus reduces to a translating task from the PIM language to the source code languages of the various platforms onto which to deploy the software. To ease the automation of such translating task, MDD proposes to divide it in two stages: first from the PIM to a Platform Specific Model (PSM) and then from such PSM to source code. The PSM is still a hybrid visual and textual notation but it incorporates concepts that are specific to one target implementation platform. The modeling languages used for the PIM and PSM must be formalized by a meta-model. The translations from PIM to PSM and PSM to source code can then be specified as transformation rules from a source meta-model pattern to a corresponding target meta-model pattern. Pattern matching is then used to generate the PSM from the PIM and the code from the PSM. In order to make this vision practical, OMG has published a set of standard languages: UML2 for the PIM, various UML profiles (i.e, UML extensions with additional domain or platform specific concepts) for the PSM, MOF2 (Meta-Object Facility) for meta-modeling and QVT (Query-View-Transformation) for model transformation. UML2 includes OCL2 (the Object Constraint Language) a formal textual language to declaratively specify first-order logical constraints on elements of UML2 models and MOF2 meta-models. OCL2 allows a PIM in UML2 to be fully refined, free of natural language and the input to full structural and behavioral automatic code generation. Most of the power of QVT is derived from its reusing OCL2 to specify transformation patterns. The latest releases of the leading UML CASE tools comply with most of these recent OMG standards and thus support MDD. The combination of these aligned standards and tools with a large industrial following is currently reviving interest in rule-based development, declarative programming, meta-programming, automatic programming and formal methods. With key contributions to all these areas, logic programming is a natural fit to fill the current MDD gaps. The logic programming community should therefore embrace MDD as a golden opportunity to make a greater impact on mainstream software engineering. The ORCAS project is an attempt to do just that.

Component-Based Development (CBD) While MDD fosters reuse of application models across platforms, CBD fosters reuse of functionalities across applications. A software component encapsulates a set of functionalities which need recurs in diverse applications. It contains metadata that specifies how to assemble these functionalities with those encapsulated in other components to build more complex ones through reuse. In an assembly, a component may act as both a server and a client. The assembly structural meta-data of a component includes provided interfaces, i.e., the signature of the operations that are available by connecting to the server ports of the component. It may also include required interfaces, i.e., the signatures of the operations which the component expects to be available in the deployment environment through connections at its client ports. A component may also include assembly behavioral meta-data that describes the pre and post conditions of the operations provided and required at its ports in terms of its own states and the states of its clients and servers in the assembly. Such meta-data allows defining a contract between a client-server component pair. Such design by contract permits black-box reuse, which is ideal for leveraging third party software and more cost-effective than the white-box reuse by inheritance in object-oriented frameworks. A component can be substituted at any time by another one that is internally different but respects the same contracts at its ports, without affecting the rest of the software. Since the word "software component" is overloaded, it is important to distinguish between a design-time component model, an implementation-time source code component, and a deployment-time bytecode or binary component. It is also important to distinguish between a design or implementation time component class that specifies a generic set of services and its potentially many run-time component instances. A source component differs from a package, module or an API in that it can be compiled and deployed independently of a larger, embedding piece of software. It must therefore include a user-interface to be usable in stand-alone mode. Binary components allow assembling together services programmed in different languages provided that they share the same deployment platform (e.g., .Net). Built-In Contract Testing (BICT) Standard testing CASE tools are monolithic, language-specific and require access to source code. Their usefulness for CBD is thus quite limited. As an alternative, BICT proposes that each component class encapsulates together with its main functionality, a companion testing facility. This facility includes one provided service testing interface accessible through a provided service testing port and a set of required service tester sub-components accessible through required service tester ports. When a tentative assembly is considered, each of its client-server relationship can then be immediately black-box tested by connecting the client's required server tester port to the server's provided service testing port. The client's server tester sub-component then generates a set of test cases and run them through this connection. Each test case makes a call to one operation of the server's provided service interface that satisfies the stipulated preconditions of this operation. It then checks whether the result returned by the call conforms to the stipulated post-condition of the operation. Assembling BICT components thus does more than generating an application through reuse. It also automatically generates a customized CASE tool for testing it. Aspect-Oriented Development (AOD) AOD uses generative techniques to separate the main functional concern of an application from orthogonal concerns related to quality of service, development and deployment such as persistence, distribution, security, fault-tolerance, debugging and testing. AOD prescribes to model and code the main concern in isolation using usual units such as components or classes. But it prescribes to model and code each orthogonal concern using a separate transformational unit called an aspect. An aspect specifies patterns (called poincuts) of locations (called join points) into the main concern model or code, around which additional model elements or code (called advices) must be inserted (or woven) to address the concern captured by the aspect. Application of these patterns results in a complete model or code that simultaneously addresses all the independently modeled or coded concerns of the application. The KobrA and ORCAS method KobrA [2] is the first software engineering method to integrate MDD with CBD and BICT. It defines a fine-grain recursive process together with a very precise list of mutually constrained artifacts to construct at each sub-step. However, it suffers from various limitations that the ORCAS method aims to overcome. First its scope is limited to the construction of the PIM. In contrast, the ORCAS method will also prescribe artifacts to use for the PSM (for selected target component platforms), as well as add steps to derive this PSM from the PIM, and then the code from this PSM. Such derivation cannot be fully automated from a KobrA PIM for it relies on UML 1.3 diagrams combined with ad-hoc tables and natural language artifacts. To overcome this limitation, ORCAS defines a UML2 profile that fully leverages OCL2 to specify an executable PIM, free from any vague natural language. Another limitation of KobrA is the lack of specific artifacts and process steps to model GUI components. In AR, such components are crucial to display the reasoning explanations in friendly-to-browse fashion. ORCAS will thus extend KobrA by incorporating recent advances in UML-based GUI design [3], [14]. ORCAS will also extend KobrA with additional artifacts and process steps to integrate AOD with MDD and CBD. At the PIM level, ORCAS separates the main concern modeled as components from the orthogonal concerns modeled as aspects. For platforms that do not support Aspect-Oriented Programming (AOP), ORCAS prescribes to weave the concerns at the PIM level. The PIM to PSM and PSM to code translation process then occurs on complete models. In contrast, for AOP platforms such as AspectJ [4] ORCAS prescribes to delay weaving until the code level. The PIM to PSM and PSM to code translations thus decomposes in parallel tracks: one track for main concern components, and one track for each orthogonal concern aspect. The last way in which ORCAS will extend KobrA is by incorporating formal methods. For AR components, this is crucial to guarantee sound reasoning. To achieve this goal, ORCAS decomposes the PIM into two levels: the knowledge level, KPIM, in UML2 and the formal level, FPIM, in CHORD, an object-oriented logical rule based constraint language. The characteristics of CHORD and their motivation are discussed in the next section.

3 CHR^v as a Versatile Formal Platform for Reasoning Services

The previous section focused on principles 1-5 of the ORCAS component framework related to its engineering method and meta-model elements. This section elaborates and motivates principles 6-10 related its library, CASE tool and target deployment platform elements. AR Services AR services can be classified along various axes. One axis is the reasoning task that is performed: monotonic deduction under Open-World Assumption (OWA), non-monotonic deduction under Closed-World Assumption (CWA), monotonic inheritance (i.e., single inheritance without overriding) and non-monotonic inheritance, belief revision with truth-maintenance, belief update (i.e., reasoning about actions, change and time), planning, abduction, symbolic or numerical constraint solving, optimization, induction and analogy. Another axis is the knowledge representation language: propositional, first-order relational, first-order object-oriented, high-order relational or high-order object-oriented. A third axis is the mathematical basis underlying the reasoning: Boolean logic, ternary logic, fuzzy logic, probability theory or decision theory. Recent research has shown that most of these reasoning tasks can be recast as other ones. Inheritance was recast as deduction [18], non-monotonic deduction under CWA and belief revision with truth maintenance as abduction [11], belief update and planning as both deduction and abduction [18]. Probabilistic deduction and abduction in Bayesian networks, as well as decision-theoretic optimization in influence diagrams were both recast as the integration of logical abduction with numerical constraint solving [8] [15]. CHR recasts symbolic and numerical constraint solving as logical deduction [8]. These results suggests a layered assembly for the ORCAS component library: monotonic logical deduction and abduction laying at the bottom level, symbolic and numerical constraint solving laying at the intermediate level and all the other reasoning services composed at the higher levels. These results also points to a CHR^v inference engine providing integrated support for deduction, abduction and constraint solving [1], as the most basic large grain component of the library.

CHR^v in a Nutshell CHR^v is a logical rule-based knowledge representation and programming language. A CHR^v program is a logical conjunction of guarded production and rewrite rules. Syntactically, the Left-Hand Side (LHS) of a CHR^v rule contains two conjunctions of First-Order Logic (FOL) terms: the propagated head and the simplified head separated by a \. The Right-Hand Side (RHS) contains a conjunction of FOL terms, the guard, and a disjunction of conjunction of FOL terms, the body, separated by a —. Each FOL term represents a constraint. CHR^v is a Turing-complete language but it is not a self-sufficient language: it necessarily delegates solving some constraints to an underlying host language or to external components. Constraints are thus divided into internally defined (or user-defined) constraints and externally defined (or built-in) constraints. Rule heads can only contain internal constraints and guards only external constraints. There are three types of rules:

- Simpagation rules of the form:
  $P_1, ..., P_i \backslash S_1, ..., S_j <=> G_1, ..., G_k | (B^1_1, ..., B^1_p); ...; (B^q_1, ..., B^q_r).$
- Simplification rules of the form:
  $S_1, ..., S_i <=> G_1, ..., G_k | (B^1_1, ..., B^1_p); ...; (B^q_1, ..., B^q_r).$
- Propagation rules of the form:
  $P_1, ..., P_i ==> G_1, ..., G_k | (B^1_1, ..., B^1_p); ...; (B^q_1, ..., B^q_r).$

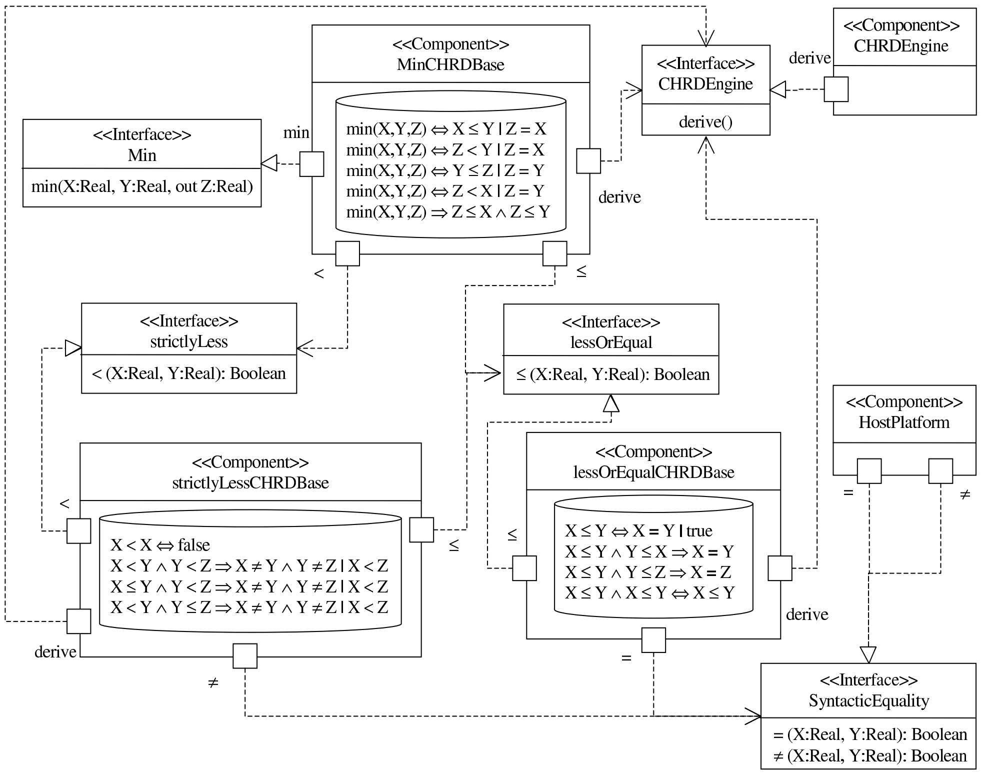
Simpagation rules extend production rules with guards and disjunctions on the RHS. Simplification rules extend conditional term rewrite rules [10] with conjunctions on the LHS and with disjunctions and conjunctions on the RHS. A simplification rule has an empty propagation head, whereas a propagation rule has an empty simplification head. A simpagation rule is syntactic sugar: it can be substituted by a semantically equivalent pair of simplification and propagation rule sharing the same RHS. If X^g_h is the set of variable occurring in the heads and guards of a rule and Y_b the set of those occurring only in its body, then the FOL declarative semantics of a propagation rule is given by the formula:
$\forall X^g_h (G_1 \wedge ... \wedge G_k \Rightarrow ((P_1 \wedge ... \wedge P_i) \Rightarrow \exists Y_b (B^1_1, ..., B^1_p) \vee ... \vee (B^q_1, ..., B^q_r)))$
Similarly, the semantics of a simplification rule is given by the formula:

$\forall X_h^g(G_1 \wedge ... \wedge G_k \Rightarrow ((S_1 \wedge ... \wedge S_i) \Rightarrow \exists Y_b(B_1^1, ..., B_p^1) \vee ... \vee (B_1^q, ..., B_r^q)))$
For simplification rules with neither guards nor conjunctions in their heads, this semantics directly corresponds to the Clarks's completion semantics of the following Prolog clauses:
$\{S_1 : -B_1^1, ..., B_p^1....S_1 : -B_1^q, ..., B_r^q.\}$ Any pure Prolog program can thus be trivially translated into a CHR^v program. Syntactically, a query to a CHR^v inference engine is a rule body. Operationally, a CHR^v engine maintains two stores: one for internal constraints and another for external constraints. Such a store is similar to the fact base of a production system or a Prolog engine, except that it may contain non-ground terms even for numerical constraints. A CHR^v engine combines guarded committed choice forward chaining with backtracking search to iteratively rewrite the initial stores that contains the query, into a consistent external store and an internal store that contains the answer to the query (or fail after having exhausted the space of alternatives). A rule fires if its heads match the internal store and its guards are entailed by the external store. Checking guard entailment is delegated to the host language. Firing a rule deletes its simplified heads from the stores. It also adds to the first alternative body of the rule to these stores. Failure triggers backtracking.

CHR^v as versatile base language for ORCAS Many reasons motivate our choice of CHR^v for the lowest layer of the ORCAS framework. First, it subsumes and combines the strengths of production rules, conditional term rewrite rules, concurrent constraint programming with guarded clauses [8] and Prolog. Second, while it uses OWA by default, propagation rules with disjunctive bodies can be used to selectively use CWA on desired constraints [1]. Third, it has also been shown to subsume some description logics [7], as well as abductive constraint logic programming [1]. It is thus a very versatile starting point. It has also been shown that association rule mining and inductive logic programming can be used to learn CHR rules from ground fact examples and background knowledge in CHR [1].

CHR^v as a Reasoning Component Assembly Platform Beyond the great versatility of CHR^v, another key factor for our choice is the distinction that CHR^v makes between internal constraint defined in the rule base and external constraint defined in the host language. It can be generalized to provide an elegant basis to assemble component-encapsulated rule bases: the constraints in the guards of a client rule base appear in the heads of a server rule base. External constraints of each client base are partitioned in N stores, one per server base. Queries about these stores are sent through the assembly connection ports. The CHR^v engine acts as a server component to all CHR^v base components. An example assembly is given in Figure 1. A CHR^v base component defines the min constraint, using two other CHR^v bases as server components: one that defines the $\leq$ constraint and one that defines the $<$ constraint. These two bases use the underlying host platform as server.

CHORD: Extending CHRV with Object-Orientation and Inheritance We intend to use CHR^v as a component assembly platform only at the lowest level of the framework. Immediately above it, assembly will be done using its object-oriented extension CHORD that we are currently designing. As mentioned in the previous section, in the ORCAS component engineering method, CHORD will serve as an intermediate formal FPIM level between the knowledge level KPIM in UML2 and the PSM level in a UML2 profile for either JavaBean, C#.Net or C++.Net. Using a purely relational language such as CHRV to mediate between two object-oriented levels would involve a cumbersome double paradigm switch for (a) component and application developers using the ORCAS method, and for (b) the ORCAS CASE tools that will aim to automatically generate the FPIM from the KPIM and then the PSM from the FPIM. CHORD aims to be to CHR^v what Frame Logic [12] [18] is to Prolog. Syntactically, it will thus extend the heads, guards and bodies of CHRV rules with Frame Logic style high-order object-oriented molecules and path expressions. Propagation rules with such molecules in the RHS will define the classes and objects. Semantically, CHORD will extend the FOL declarative semantics of CHRV rules with FOL semantics for multiple, single-source inheritance. Defining this unified semantics is the main theoretical challenge of the ORCAS project. Once available, this semantics will serve as the basis to verify key properties of a KPIM prior to the generation of the PSM, using a CHORD inference engine.



<<Component>> MinCHRDBase
min(X,Y,Z) ⇔ X ≤ Y | Z = X
min(X,Y,Z) ⇔ Z < Y | Z = X
min(X,Y,Z) ⇔ Y ≤ Z | Z = Y
min(X,Y,Z) ⇔ Z < X | Z = Y
min(X,Y,Z) ⇒ Z ≤ X ∧ Z ≤ Y
<<Interface>> Min
min(X:Real, Y:Real, out Z:Real)
min
derive
<
≤
<<Interface>> CHRDEngine
derive()
<<Component>> CHRDEngine
derive
<<Interface>> strictlyLess
< (X:Real, Y:Real): Boolean
<<Interface>> lessOrEqual
≤ (X:Real, Y:Real): Boolean
<<Component>> HostPlatform
=
≠
<<Component>> strictlyLessCHRDBase
X < X ⇔ false
X < Y ∧ Y < Z ⇒ X ≠ Y ∧ Y ≠ Z | X < Z
X ≤ Y ∧ Y < Z ⇒ X ≠ Y ∧ Y ≠ Z | X < Z
X < Y ∧ Y ≤ Z ⇒ X ≠ Y ∧ Y ≠ Z | X < Z
<
≤
derive
≠
<<Component>> lessOrEqualCHRDBase
X ≤ Y ⇔ X = Y | true
X ≤ Y ∧ Y ≤ X ⇒ X = Y
X ≤ Y ∧ Y ≤ Z ⇒ X = Z
X ≤ Y ∧ X ≤ Y ⇔ X ≤ Y
≤
derive
=
<<Interface>> SyntacticEquality
= (X:Real, Y:Real): Boolean
≠ (X:Real, Y:Real): Boolean

Fig. 1. CHR-Based Automated Reasoning Component Assembly.

4 Current Status and Roadmap for next Steps

Our current efforts span four parallel tracks. The first is the definition of a UML2 profile for the KPIM. The current version already covers artifacts for MDD, CBD and AOD. Extensions for BICT and GUI are under way. The second is the definition of CHORD for the FPIM. A preliminary version of its abstract syntax is available as a MOF2 meta-model. The third is the definition of a UML2 profile for the PSM. We are currently elaborating ones for Java Beans and C++. The fourth is the development of a CHRV engine. We have extreme programmed in AspectJ and Java such an engine called CHREK. It is the first available CHRV engine running on a mainstream IT platform that can be straightforwardly integrated inside a variety of applications. We are now reversing engineering CHREK code into an executable UML2 KPIM. From this KPIM, we will then be able to apply the ORCAS method to incrementally refactor and extend CHREK into the first complete component of the ORCAS library.

5 Conclusion

In this paper, we overviewed the long-term project ORCAS that aims to overcome the current difficulties that one faces to understand, learn to use, integrate in practical applications, customize, extend or reuse AR software. We proposed ten design principles to achieve this goal. Using the ORCAS framework, inference engines, domain knowledge bases, automated reasoning services based on such engines and bases, and finally applications using these services will all be assembled from components, some new, some provided by the framework. Using the ORCAS method, each component is first modeled down to the execution granularity in UML2. This resulting KPIM is translated into an FPIM in CHORD, a Frame Logic based object-oriented extension of CHRV.

Selected properties of this FPIM can then be formally verified using a CHORD inference engine. Once verified, this FIM get translated into a PSM using a UML2 profile for either Java or .Net. Aspect-oriented and object-oriented code is then generated from this PSM, before being woven, compiled and deployed as a bytecode or binary component, ready to be assembled in larger systems.

References

1. Abdennadher, S. Rule-based Constraint Programming: Theory and Practice, Habilitation, Institut fr Informatik, Ludwig-Maximilians-Universitt Mnchen, 2001.
2. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J. and Zettel, J. Component-based Product Line Engineering with UML. Addison-Wesley, 2002.
3. Blankenhorn, K. A UML Profile for GUI Layout. Master's Thesis. University of Applied Science Furtwangen, 2004.
4. Clarke, S. and Baniassad, E. Aspect-Oriented Analysis and Design : The Theme Approach. Addison-Wesley. 2005.
5. Costa, V., Page, D., Qazi, M. and Cussens, J. Constraint Logic Programming for Probabilistic Knowledge. In Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI'03), Acapulco, Mexico, 2003.
6. Eriksson, H.E, Penker, M., Lyons B. and Fado, D. UML 2 Toolkit. OMG Press. Wiley. 2004.
7. Frühwirth, T. The Web CHR site. http://bach.informatik.uni-ulm.de/ webchr/
8. Frühwirth, T. and Abdennadher, S. Essentials of Constraint Programming. Springer. 2003.
9. Gross, H.B. Component-Based Software Testing with UML. Springer. 2004.
10. Joannaud, J.P. Rewrite Proofs and Computations. In Computer and Systems Sciences 139, SpringerVerlag, 1995.
11. Kakas, A.C., Kowalski, R.A. and Toni, F. The Role of Abduction in Logic Programming. 1998.
12. Kifer, M., Lausen, G. and Wu, J . Logical Foundations of Object-Oriented and Frame-Based Languages. Journal of the ACM 42. 1995.
13. Kleppe, A., Warmer, J. and Bast, W. MDA Explained - The Model-Driven Architecture: Practice and Promise. Addison-Wesley. 2003.
14. Koch, N. Software Engineering for Adaptive Hypermedia Systems: Reference Model, Modeling Techniques and Development Process. PhD. Thesis, Ludwig-Maximilians-Universitt Mnchen, 2000.
15. Poole, D. Logic, Knowledge Representation and Bayesian Decision Theory. Proceedings of the First International Conference on Computational Logic (CL'2000). 2000.
16. Russell, S. and Norvig, P. Artificial Intelligence: A Modern Approach. 2nd Ed. Prentice Hall. 2003.
17. Shanahan, M.P. An Abductive Event Calculus Planner. Journal of Logic Programming 44, 2000.
18. Yang, G. A Model Theory for Nonmonotonic Multiple Value and Code Inheritance in Object-Oriented Knowledge Bases. PhD. Thesis, Computer Science Department, Stony Brook University of New York, 2002.