Architecture and Functionality of a Specification Environment for Distributed Software

Bernd Krämer

Computer Science Department, Naval Postgraduate School
Monterey, CA 93943

International Computer Science Institute
Berkeley, CA 94704

Abstract

One of the key factors in the acceptance of formal methods in software development is the availability of extensive automated support. Graspin is a workstation-based prototype environment that aids in the incremental construction, verification, and prototyping of specifications for concurrent and distributed software systems. It includes a Petri net-based specification formalism, an editor generator with graphical capabilities, tools for static semantics checking, for automated verification of static and dynamic properties of specifications, and for specification-based prototyping. This paper addresses the main functions and design of Graspin and sketches methodological issues of how its tools are used to develop, test, and verify formal specifications.

1 Introduction

The significance of formal specifications in the process of software development has been widely recognized [18]. High technology initiatives such as the Alvey Program or ESPRIT have supported various joint efforts among industry and academia aiming at an improvement of formal methods for use in large scale practical projects and the development of computer-aided support tools. These efforts have led to a better understanding of the capabilities of formal specification methods and created some evidence that formal methods could be a response to the increasing complexity and quality requirements software development industry is confronted with [12]. Some of the better known specification methods such as VDM [2], OBJ2 [6], Z [23] and others have already been used in industry.

The goal of the work reported here was to automate the design, test, and verification of formal and visual specifications of concurrent and distributed software through a workstation-based integrated environment, Graspin1. The driving component underlying the implementation of Graspin is a semi-graphical specification formalism, S&RS2. The formalism is based on a well-engineered integration of high-level Petri nets and abstract data types. It was designed by the authors as an attempt to overcome the limitations of current specification methods to adequately cope with fundamental issues of distributed systems, such as concurrency, nondeterminism, synchronization, and communication.

This paper is organized as follows: In the next section the Graspin specification language is sketched. Design and main functions of Graspin tools are described in Section 3. In Section 4 some of these functions are illustrated by tracing aspects of the stepwise construction and validation of a concurrent mergesort system. Section 6 discusses related work.

2 Language Overview

A S&RS specification describes a distributed system in terms of abstract data types and a collection of dynamic subsystems which can exhibit concurrent behavior. The data structure on which a system operates is specified as an abstract data type using the algebraic sublanguage of S&RS. Dynamic system behavior is specified by means of high-level Petri-nets, called Σ-nets. They are formally defined in [15]. Each net defines an "interaction protocol" in terms of the system component's relevant states and transitions.

The algebraic sublanguage features a syntactic distinction between total and partial functions, universally quantified conditional equations for specifying semantic properties of functions, and an elaborate type system. The latter supports parametric, function, and dependent types to increase the expressive power of the language, while preserving the advantages of static typing [16]. These type concepts make specifications more concise and provide generic capabilities which mimic simple cases of parameterized specifications [4].

The example in Fig. 1 illustrates the S&RS notation. It presents a generic specification of a list data type. Function declarations are written in a Pascal-like style. The colon in con-

---

1 This project was supported in part by the Commission of the European Communities within the ESPRIT project 125 Graspin and was performed while the authors were with GMD.

2 S&RS is a registered trademark of GMD.
ditional equations separates the conclusion on the left-hand side from the premises on the right-hand side. Expressions of the form \( t \) represent equations with equal sides (\( t = t \)). Such equations require the definedness of \( t \) when being a premise or conclude the definedness of \( t \) when occurring as a conclusion.

```
data generic-lists provides
type list (T:types).
fun nil: list (T).
  cons (E:T, L: list (T)): list (T).
  nth-first, nth-rest (N:int, L: list (T)): list (T).
  map (F:T1 \rightarrow T2, L: list (T)): list (T2).
  flatten (L: list (list (T))): list (T).
var T, T1: types, E: T, L: list (T), F: T \rightarrow T1.

seq nth-first (N, nil) = nil.
  nth-first (N, cons (E, L)) = cons (E, nth-first (N - 1, L));
  nth-rest (N, nil) = nil.
  nth-rest (N, cons (E, L)) = L.
  nth-first (N, cons (E, L)) = nth-rest (N - 1, L);
  nth-rest (N, nil) = nil.
  nth-rest (N, cons (E, L)) = nth-rest (N - 1, L).
  map (F, nil) = nil.
  map (F, cons (E, L)) = cons (F (E), map (F, L)).
end
```

Figure 1: A generic data type specification

Data specifications are semantically based on many-sorted partial algebras as defined in [21]. In general, a data specification as shown in Fig. 1 represents an infinite first-order specification whose initial algebra semantics is taken as semantics of the data specification. The set of sorts \( S_0 \), operation symbols \( \Sigma \), variables \( X \), and conditional equations \( EQ \) of the first order specification underlying a well-typed data specification are generated by all consistent substitutions of ground terms for variables in type expressions [14].

Fig. 2 shows a screen dump of the Graspin workstation with a specification of a concurrent mergesort system being under construction. The graphical window depicts the dynamic behavior of the mergesort system using the well-known graphical form of Petri nets. The text window on the left contains the specification of functions and variables used in the net. Functions \( h \) and \( r \) serve to concurrently split a list of numbers into two parts, while function \( m \) is needed to aggregate the sorted list from sorted sublists. This specification builds on a specialization of the list data type to numbers.

In an \( S \)-nets each State element (S-element) \( a \) is labeled with a string \( u \) of sorts which defines the type of data tokens admissible as markings of \( a \). Markings describe distributed system states. They associate a set \( \{ a_1, \ldots, a_m \} \) of \( m \)-tuples of terms \( a_i = (a_{i1}, \ldots, a_{in}) \) with each S-element such that each \( a_{ij} \) is of sort \( v_j \). Each Transition element (T-element) \( b \) is labeled with a set \( \{ X_1, \ldots, X_k \} \) of variables and is constrained by a set \( \{ e_1, \ldots, e_l \} \) of equations. All variables in the constraint of \( b \) must occur in the labeling of \( b \). Labeled T-elements are schemes for concurrent elementary state changes, called events. Each substitution \( a = \{ x_1 / X_1, \ldots, x_k / X_k \} \) of variables in the labeling of

![Figure 2: Screen dump of a Graspin session](image-url)
b by ground terms \( n \) defines a particular event \((b, o)\). The constraint associated with b restricts the set of derivable events, depending on the structure of data in the marking of input S-elements of b.

A labeled and marked S-element and a labeled and constrained T-element are graphically represented by

\[ \begin{array}{c}
        \text{b} \\
        (a, b, c) \\
        \text{a}
\end{array} \]

Each flow element \((a, b)\) of a \( \Sigma \)-net is labeled with a set \( \{t_1, \ldots, t_q\} \) of n-tuples \( t_j = (t_{j1}, \ldots, t_{jn}) \), where each \( t_{jn} \) is of sort \( w \). Moreover, all variables in the tuples \( t_j \) must be contained in the labeling of the adjacent T-element. The \( t_j \) schematically denote data tokens that are removed from the marking of adjacent input S-elements and are added to adjacent output S-elements as events occur. The data tokens involved in an event \((b, o)\) are determined by applying \( o \) to the \( t_j \) in the labeling of flow elements connected to \( b \). Flow elements are represented by

\[ \begin{array}{c}
        \text{a} \\
        (c, \ldots, d) \\
        \text{c}
\end{array} \]

The semantics of a system specification is defined by the initial algebra providing the semantics of the data types it includes and a Condition-Event system\(^2\) which is uniquely determined by the \( \Sigma \)-net and the algebra. A more abstract semantics, which is consistent with the Condition-Event systems semantics, is presented in [15].

Specifications of large distributed systems can be constructed incrementally by composing them from smaller specifications using standard operations on specifications such as extension, combination, or renaming. In SEDAS these operations were appropriately extended to behavior specifications.

3 Support Tools

As we initially designed the Graspin environment we faced two major challenges: integration and extensibility. Integration refers to the ability of supported methods to enable smooth transitions among them and of environment tools to cooperate on the basis of common and uniform interfaces, style of operation, and communication medium. Extensibility refers to the ease with which an operational environment can be extended by new methods and tools that were not foreseen in the initial design.

3.1 The Graspin Kernel

The technical solutions Graspin offers to meet these challenges are comprised in the Graspin kernel which is common to all Graspin-based environments [17]. It includes

- language definition facilities based on an open-ended meta formalism for specifying environment tools, ASDL;
- an object-oriented internal representation of software objects with automatically generated manipulation operations for object creation, modification, and inspection;
- a common object base;
- generic editing tools with graphical capabilities.

ASDL extends the power of similar meta languages based on context-free grammars such as MetaL [13] or SSL [19] by multiple inheritance and syntax-directed translation schemes. Translation schemes are implemented as generic functions on objects. In the current implementation ASDL is linked to Common Lisp as foreign language, which means that Lisp functions and data types can be used in an ASDL specification. Foreign functions may perform simple computations on Lisp data types or may implement complex algorithms of foreign tool to be intergrated in the kernel environment.

Language definitions in ASDL are compiled into an object-oriented representation providing functions to create, decompose, modify and inspect objects representing software fragments written in the specified language. Translation schemes allow declarative specifications of complex structure-driven computations and object transformations. They are implemented as methods and are equipped with method combination facilities to support change and reuse. All these operations encapsulate an object base which serves as a common communication medium among the environment tools.

3.2 Hybrid Editing

The screen dump in Fig. 2 shows how Graspin supports the semi-graphical notation of SEDAS through a hybrid structure editor. The standard user interface of Graspin uses an enhanced version of Emacs for free-style editing of textual specification fragments. Such fragments are submitted to an incremental syntax-directed parser for immediate syntax check and storage in the environment base, provided they are syntactically correct. Conversely, an incremental pretty printer unparses the internal representation of objects into a formatted text. The user interface also provides Graspin graphics windows with mouse and menu interaction for graphical editing of \( \Sigma \)-nets, elementary forms of Petri nets, graphs, and diagrams. A graphical refinement feature is available for all diagrammatic notations supported.

All Graspin functions can be invoked directly from the editor or, via menu selection, from Graspin windows (see the menu pane in Fig. 2 for the top level menu).

3.3 Polymorphic Type Checking

Graspin includes a SEDAS type checker that builds on Robinson unification to infer substitution instances of parametric and dependent types. It implements the static type checking algorithm described in [16]. The type checker is completely specified in ASDL. It verifies the consistency between declaration and applications of operation and type symbols. As declarations of symbols involve terms denoting parametric types, the declaration of a symbol must be type checked before its applications can be type checked. This leads to a hierarchical type checking scheme in which symbols that use only built-in types (type, nat, unit) are type checked first.
Before verifying type consistency, an identification step is performed. It includes checks of simple context conditions and the flattening of structured specifications.

3.4 Execution Support and Algebraic Verification

The central tool to support formal analysis of data specifications, reasoning about desirable properties of SDEAS specifications, and testing of specification through symbolic execution is a rewrite rule subsystem (RRS). It applies Knuth-Bendix completion when attempting to generate a complete confluent and terminating rule system from an equational specification. It checks the consistency and completeness of the rule system and verifies the totality of operations (cf. the Graspin verification window underneath the graphical window in Fig. 2). If a specification extends another specification, the system checks that the combination of the two neither produces confusion nor junk. For these checks RRS relies on a distinction between constructor and selector operations. To improve the efficiency of term rewriting and to reduce storage requirements, a rule system can be compiled into Common Lisp functions.

The main function of RRS is symbolic execution of specifications. It evaluates operations by rewriting terms into their normal forms. It also verifies the constraints associated with transitions while checking their activation conditions. RRS also supports inductive proofs by simplifying terms and verifying equivalences. However, it is not an automatic theorem prover. Some steps in the construction of proofs have to be done by hand. Such proof techniques have been demonstrated in detail in [18] following the approach of [8].

3.5 Net Animation

The graphics editor provides functions to produce dynamic behaviors of the intended system through net animation. It is a specific form of net simulation in which event occurrences are visualized by redistributing data tokens on the S-elements of a net using the drawing functions of the graphics editor.

The simulation function directly interprets a given net and uses RRS functions for rewriting data tokens and locally evaluating transition constraints. It can be characterized as a 'Match-Assign-Reduce-Change' method, short 'MARC'. Since there are, in general, infinitely many possible substitutions for the variables in the labeling of a T-element, substitutions are derived from the markings of input S-elements of a T-element. To make this feasible, MARC requires that substitutions of certain variables in the labeling of input arcs are sufficient to determine the values of all other variables (on input and output arcs) by rewriting the transition constraints. In the first step, matching, a set of tokens is selected from the input places of some transition. These tokens must must match the input constructor terms. The tokens define ground term substitutions for the variables in input terms. The substitutions are assigned to the variables and are used for reducing the equations in the transition constraint to ground equations and for reducing all remaining arc terms to ground terms. At this point the activation of an event can be checked and the state change is eventually carried out. Is the net in Fig. 2, for example, variables P1 and L1 on the incoming arc of transition aggregate together with the equations of aggregate fully determine the substitutions for the other variables of that transition.

A simulation protocol can optionally be generated. This protocol records the partial order of event occurrences together with their pre- and post-conditions.

3.6 Liveness and Safeness Analysis

Another interesting analysis capability of Graspin allows the designer to verify that system specifications have certain behavioral properties. In the context of concurrent and distributed systems liveness and safeness properties are of major concern. We distinguish a weak and a strong form of liveness: deadlock freedom guarantees that there is always an event that can occur, in other words, the system always can do something; strong liveness guarantees that each transition can eventually happen again. Safeness ensures that no overflow situations may occur.

Partly such behavioral properties can be proven using net simulation. For instance, simulation may reveal dead states, where no transition can occur, or home states, i.e., reproducible states. A home state implies that a certain subnet is strongly live because we always can roll back to the home state and retry the operations again. In general however, simulation has the character of testing only. This means, it is capable of showing the presence of bugs but not capable of proving their absence, such as the absence of deadlocks. Analysis of liveness and safeness is based on a decomposition theorem of so-called free-choice like \( \Sigma \)-nets [21]. These nets have the special property that they are decomposable into simpler nets representing sequential subsystems. Liveness and safeness analysis of \( \Sigma \)-nets has been studied extensively in [22].

4 Stepwise Development using Graspin

Having constructed a SDEAS specification with the hybrid editor, useful next steps would be to type check it and try to construct a rule system for it. Thereby it may happen that the specification passes the type checking process but is rejected by RRS. This situation occurs if a data specification involves conditional equations with non-empty premisses, partial functions, or parametric types because these features are not handled by the current implementation of RRS. To circumvent this restriction, the designer has to transform conditional equations into (less readable) unconditional ones using an if-then-else-function; if possible, he has to replace partial functions by boolean-valued total functions, and has to eliminate types. Clearly, the first two steps cannot always be achieved as both conditional and partial specifications are more powerful than pure equational specifications with totally defined functions.

4.1 Visualizing Dynamic Behavior

Having generated a rule system, the designer might want to visualize the dynamic behavior of a system specification for selected input data. For example, the concurrent-merge sort system could...
be animated on the following list example

\[[14, 6, 57, 13, 11, 9, 4, 23, 3],\]

using a Prolog like abbreviation for lists. (Such representations can be easily implemented by simple Lisp functions that can be associated with T-elements.) Fig. 3 shows an intermediate and the final state of the net simulation for this example.

The designer has various choices to control the simulation process, e.g., by selecting whether maximal steps of concurrent transitions, randomly selected subsets, or singleton change steps are performed. Moreover he has various choices concerning the indication of simulation information such as the set of substitutions determined by the activation check, the indication of activated transitions, or the time interval taken to present such information.

![Figure 3: Distributed states of computation](image)

**4.2 Inductive Proofs**

To gain further security about a specification we can also formulate correctness criteria and attempt to prove them using RRS functions. A well-known proof technique for abstract data types is structural induction over its constructor operations. This technique can be used to verify the following property of the splitting operations \(lh, rh\) defined in concurrent-mergesort:

\[
\text{append}(lh(L), rh(L)) = L
\]

To perform this proof, we have to show that the above theorem holds for the base case, i.e. for \(L = \text{nill}\). This is easy to show because \(lh(\text{nill})\) and \(rh(\text{nill})\) immediately rewrite into their normal form \text{nill}. Using the results of [8] we then can extend our specification by new constant symbols and an equation:

\[
\text{fun} \ l : \text{list} \Rightarrow \text{nati}.
\]

\[
\text{eqn} lh(l) \& rh(l) = 1
\]

which we use as assumption for the induction step. Extension checking automatically ensures that the extended specification preserves the semantics of the original list specification. The induction step then consists in showing that:

\[
\text{append}(lh(\text{cons}(n, 1)), rh(\text{cons}(n, 1))) = \text{cons}(n, 1)
\]

holds. In the proof we have to consider two cases: a) if \(\text{length}(1)\) is even, we can deduce from properties of \(lh\) and \(rh\), which have to be verified by additional lemmas, that \(lh(\text{cons}(n, 1)) = \text{cons}(\text{lh}(1))\) and \(rh(\text{cons}(n, 1)) = \text{rh}(1)\) so that \(\text{cons}(n, 1)\) follows from the definition of \(\text{append}\); b) for the odd case the proof is a little more cumbersome but is basically conducted in the same way using some additional lemmas (cf. [15] for further details).

**4.3 State Invariants and Termination**

An important property of the dynamic behavior we would also like to verify is an invariant stating that for any possible marking of the net the union of all distributed sublists is a permutation of the list in the initial marking. We use a technique which combines induction over the set of reachable markings with structural induction over the data token involved.

To prove the correctness of the behavior specification we further have to show that, whenever computation terminates, i.e., no transition is activated, the list contained in the final marking is a sorted permutation of the initial list. For such proofs the same arguments as before apply. Again RRS simplification and equational deduction can be used to support termination arguments.

In fact, using this proof technique we detected an error in the behavior specification in Fig. 2 which did not show up during animation: in the trivial case of the initial marking with an empty list \text{nill}, neither distribute nor process are activated because their constraints require that \(\text{length}(L) > 1\) or equals 1, respectively, but actually \(\text{length}(\text{nill}) = 0\). To correct this bug, we have to replace the equation associated with transition \text{process} by the equation \(\text{?length}(L) \leq 1\).

**5 Current Status**

Graspin has been implemented on Symbolics 3600 Lisp machines in Common Lisp and the Lisp-based meta language ASDL using the PCL implementation of CLOS (Common Lisp Object System). The environment was later ported to Sun workstations using X11 via the public domain implementations of CLX (Common Lisp access to X) and CLUE (Common Lisp User Interface Environment). In the current state, however, this implementation suffers from severe performance lacks which are mainly due to the many levels through which the user interface is accessed and partly due to still insufficient support of CLOS on these workstations.

**6 Related Work**

There is a number of other support environments available which rely on algebraic specification methods. Some of them ease the construction of formal specifications by tools which implement design strategies based on semantic knowledge about the formalism (e.g., [1]); others focus on rapid-prototyping issues based on
rewrite rule techniques (e.g., [11, 9, 7]) or emphasize computer aided transformations of specifications into efficient code (e.g., the CIP system [3]). But unlike Graspin, none of these tools supports symbolic execution of concurrent behavior specifications.

There is also a number of Petri net tools around which influenced the design and functionality of Graspin and provide illustrative material for future extensions. An exhaustive survey on Petri net tools is given in [5]. Most of the net tools, however, support only the basic class of Petri nets, so that Graspin seems to be the first environment to support the execution of fully abstract high-level Petri nets.

7 Conclusions

The Graspin architecture and kernel environment have shown their flexibility in the development of a prototype environment supporting the formal specification language SDRAS. The work reported here made it necessary to extend the kernel by semantic tools such as type checker and net simulator and to integrate separately developed tools such as the RBS into a coherent environment. New Graspin features that were operational as this article was revised include a graphical refinement method and a multi-level net animation technique.

Our experiences with this prototype development confirmed the expected properties of ASDL and its implementation, namely to support rapid prototyping of new environment tools and to ease incremental change and reuse. The Graspin prototype has been successfully used in several experiments including the specification and partial verification of parallel sorting algorithms and electronic retail payment systems, the design and verification of programmable logic controllers [10], and the implementation of an experimental object-oriented simulator for artificial neural networks [20].

References