Integrating Action Definitions into Concurrent Declarative Programming

Rachid Echahed and Wendelin Serwe

Laboratoire LEIBNIZ – Institut IMAG, CNRS
46, avenue Felix Viallet, F-38031 Grenoble, France
Tel: (+33) 4 76 57 48 91; Fax: (+33) 4 76 57 46 02
`Rachid.Echahed@imag.fr` `Wendelin.Serwe@imag.fr`

Abstract. The semantics of a process defines the actions it can execute. Thus actions constitute an essential notion for concurrent systems. In this paper, we tackle the problem of integrating the definition of actions within the context of concurrent constraint functional logic programming. We propose to define actions in a meta-language, where abstract data types of programs are available, as functions from constraint functional logic programs to themselves. We illustrate our approach through some examples and compare our approach to related work.

1 Introduction

A purely declarative program can be seen as a static theory presentation, which can be used for reasoning over the underlying model, e.g., by solving goals as in the logic programming paradigm or by simplifying expressions as in the functional programming paradigm. However, due to their "static" nature, the concepts underlying declarative languages are not sufficient to capture the whole complexity of real-world applications [29], which require coordination of concurrent activities as well as interaction with the environment. For instance, a program which uses a constant *date* as a value has to update it regularly, as well as a database recording the daily evolution of bank accounts has to evolve along with the operations executed on the accounts. Even more obvious is the case of interactive applications, since the interactions with the environment (e.g., other systems or users) have naturally an impact on the theory.

So, applications of this kind are required to modify theories, i.e., declarative programs. Most existing approaches for concurrent extensions of declarative languages, as for instance [3, 6, 7, 17, 24, 25, 27], provide actions in form of several built-in primitives, dedicated to update current theories, e.g., the "predicates" `assert` and `retract` of Prolog, the action tell of ccp, the assignment "functions" (with *effects*) `setq` (respectively, :=) of Common Lisp and Scheme (respectively, SML), or the built-in actions for changing the code of a module of Erlang. But unfortunately, there is no standard set of such actions, and each language suggests its own ones.

In this paper, we advocate that considering declarative programs as data objects in a metalanguage allows to subsume the different existing approaches for building actions over programs. If we consider (declarative) programs as elements of an abstract data type (ADT), all the different modifications can be considered as operations defined in the metalanguage on this ADT. This has several advantages. First, the approach for the integration of concurrency and mobility becomes more abstract and generic and can be applied to any declarative language, since it does not rely on special built-in primitives which might be difficult to define for other languages. Thus we gain the same advantages as when passing from data definitions to abstract data types. Second, the programmer may be given the possibility to define its own, application specific actions. This should lead to a better structured and in consequence more readable and maintainable code. Last, but not least, the ADT of programs can be used as a kind of "glue language" which permits the interaction between several components: In fact, to manipulate other components we need to know their precise structure. Thus exhibiting the ADT of programs defines the "interface" to the language (and consequently components written using the language) more formally.

The rest of the paper is organized as follows: The next section reviews briefly our framework for concurrent declarative programming. Section 3 introduces the definition of actions and presents an example of an ADT of programs, which is used to illustrate some definitions of actions in section 4. A comparison with some related work is subject of section 5. Finally, section 6 concludes.

2 A Concurrent Declarative Programming Framework

In this section, we briefly review the broad outlines of our proposal for a multi-paradigm framework combining mobile processes and declarative programming. We will not present the framework in all its aspects but rather focus on the parts we will need in this paper. The interested reader may consider [12, 13] for further information. We assume the reader is familiar with classical notions of rewriting [10, 19] as well as process algebra [4].

In our framework, an application (or system) is modeled as a set of components. Loosely speaking, a component will consist of two parts $\mathcal{C} = \langle F, P \rangle$, where P is a set of process definitions and F is a set of formulæ describing a traditional declarative program, called a *store* in the sequel.

The execution model of a component is similar to the one of concurrent constraint programming (ccp) [26] or the coordination language Linda [14] and can be characterized as in Fig. 1. Processes (p_i) communicate by modifying the common store F, i.e., by altering, in a non-monotonic way, the current theory described by the store, for example by simply redefining constants (e.g., adding a message in a queue) or by adding or deleting formulæ in F. Every change of the store is the result of the execution of an *action.* Thus actions constitute the basic entities for building processes. Interaction with the environment is based on the same mechanism: processes are allowed to execute actions on the stores of other components. Examples of actions are $\langle F, (\mathsf{tell}\ (\mathit{stick}\ 4)) \rangle$ which adds the atom $(\mathit{stick}\ 4)$ to the store F, $\langle s, c := v \rangle$, to change the definition of the (variable) constant c (which is defined in the store of the component s) to the

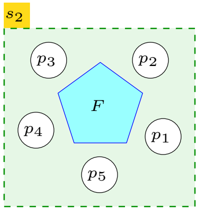

Fig. 1. execution model

new value v, $\langle Window, \mathsf{close}\rangle$ to close the window $Window$, $\langle Engine, \mathsf{stop}\rangle$ to stop the engine $Engine$ and $\langle Display, (\mathsf{print}\ \texttt{'hello'})\rangle$ to print the string `hello` on the display $Display$.

Orthogonally to the execution of actions, the store can be used as usual for a declarative program, i.e., for goal solving or the evaluation of expressions.

2.1 Stores

A store is a declarative program, presented as usual by a signature Σ declaring the different symbols used in the program and a collection of judgements $\mathcal{R}$ defining the meaning (or semantics) of the symbols. To simplify the presentation, and without loss of generality, we choose as an example of a declarative language for this paper a language based on conditional rewrite rules, and we follow a constructor-based discipline. Note that the concepts presented in this paper also apply to more realistic, i.e., more sophisticated, languages (using constraints, definitional trees, etc.).

Definition 1. *A* signature *Σ is a pair $\langle S, \Omega\rangle$ where S is a set of* sorts *(including Bool, the sort of boolean values) and Ω is a family of sorted* operations*.*

A conditional rewrite rule *is a triple "$\mathsf{lhs} \to \mathsf{rhs} \mid \mathsf{cond}$", where* lhs *and* rhs[1] *are terms and* cond *is a boolean valued term.*

Example 1. Using the signature $\Sigma = \langle S = \{Bool; Nat\};\ \Omega = \{succ : Nat \to Nat; zero\colon Nat; \mathrm{true}\colon Bool; +, -, \mathrm{mod}\colon Nat \times Nat \to Nat;\ <, \geq\colon Nat \times Nat \to Bool\}\rangle$ we can define the operation "mod" by the following two rules:

$$x \ \mathrm{mod}\ y \to x \qquad \mid x < y \tag{1}$$
$$x \ \mathrm{mod}\ y \to (x - y)\ \mathrm{mod}\ y \quad \mid x \geq y \tag{2}$$

2.2 Processes

Besides the definition of operations, a component may also include process definitions in order to deal with the dynamics of real-world applications. Processes in our framework are defined in the style of a process algebra (see, e.g., [4]).

[1] lhs (respectively, rhs) stands for "left (respectively, right) hand side".

Definition 2. *A* process term p *is a well-typed expression defined by the following grammar:*

$$p ::= \mathsf{success} \;\big|\; \big[g \Rightarrow \big\langle s_i,\ \mathsf{a}_i\big\rangle_i\big] \;\big|\; (\mathrm{q}\ t_1\ \ldots\ t_n) \;\big|\; p\,;p \;\big|\; p \parallel p \;\big|\; p+p \;\big|\; p \oplus p$$

The process $\mathsf{success}$ represents the process which terminates successfully. A guarded action $\mathsf{a} = \big[g \Rightarrow \big\langle s_i,\ \mathsf{a}_i\big\rangle_i\big]$ is composed of a *guard* (i.e., a boolean expression) g and a sequence of pairs of *(elementary) actions* a_i and *storenames* s_i. Informally, a storename can be seen as a (symbolic) identifier for a component, by denoting the store of the component. So, executing a guarded action means to test the validity of the guard in the (local) store (i.e., the guard may be reduced to true), and, upon the validity of the guard, to execute (atomically) the sequence of the (elementary) actions[2] $\langle s_i, \mathsf{a}_i\rangle$. The execution of the (elementary) action $\langle s, \mathsf{a}\rangle$ replaces the store denoted by the storename s, say F, by the result of the application of the (elementary) action a to it, i.e., $(\mathsf{a}\ F)$. The definition of (elementary) actions will be presented in more detail in section 3. $(\mathrm{q}\ t_1\ \ldots\ t_n)$ stands for a call to the process q. The definition of processes is given shortly below.

As usual in process algebra, we use some operators for combining processes: parallel ($\parallel$) and sequential (;) composition, nondeterministic choice (+) and choice with priority ($\oplus$). The last operator is not very common, but we found it necessary to model critical applications where nondeterminism is not acceptable [1]. The intended meaning of the process term $p_1 \oplus p_2$ is: "execute the process p_2 only if the process p_1 cannot be executed now", i.e., the process p_1 has a higher priority than the process p_2.

Process definitions are intended to give a description of the behaviour of processes. Some restrictions are required on the (recursive) definition of processes in order to avoid pathologic cases, especially processes with an infinite branching degree. A common solution to avoid such problems consists in requiring process definitions to be *guarded*.

Definition 3. *A* process q *is defined by a judgement of the form*

$$(\mathrm{q}\ x_1\ \ldots\ x_n) \Leftarrow \textstyle\bigoplus_{j=1}^{m} \mathsf{a}_j\,;p_j$$

where (for each j) a_j is a guarded action and p_j is a process term. For a readable presentation, we omit here some formal technical conditions on the use of variables.

According to definition 3 a process is defined by a set (ordered by priority) of "bodies", each of which consist of a (guarded) action and a process term.

Example 2. Consider the problem of the "Dining Philosophers" [11]. We model the situation with two boolean functions, which have to be added to the signature of example 1, namely (stick x) and (is_eating y). The former represents the fact

[2] Abusing the notation, we call a pair ⟨storename, (elementary) action⟩ for short (elementary) action.

$$
\begin{array}{lll}
(\text{thinks } x\ n) \Leftarrow \left[\begin{array}{l}(\text{stick } x) \quad\quad \wedge \\ (\text{stick } ((x{+}1) \bmod n))\end{array} \Rightarrow \begin{array}{l}\langle F,\ (\mathsf{del}\ (\text{stick } x))\rangle; \\ \langle F,\ (\mathsf{del}\ (\text{stick } ((x{+}1) \bmod n)))\rangle; \\ \langle F,\ (\mathsf{tell}\ (\text{is\_eating } x))\rangle\end{array}\right]; & (\text{eats } x\ n) \\
(\text{eats } x\ n) \quad\ \Leftarrow \left[\text{true} \Rightarrow \begin{array}{l}\langle F,\ (\mathsf{del}\ (\text{is\_eating } x))\rangle; \\ \langle F,\ (\mathsf{tell}\ (\text{stick } x))\rangle; \\ \langle F,\ (\mathsf{tell}\ (\text{stick } ((x{+}1) \bmod n)))\rangle\end{array}\right]; & (\text{thinks } x\ n)
\end{array}
$$

Fig. 2. process definitions for the Dining Philosophers (located on store F)

that stick x is lying on the table, and the latter is true whenever philosopher y is eating.

Using the (elementary) actions ($\mathsf{tell}\ r$) (respectively, ($\mathsf{del}\ r$)) which add (respectively, remove) an (unconditional) rule "$r \rightarrow$ true" to (respectively, from) the store F, i.e., the declarative program, we can model the behavior of a philosopher by the processes of Fig. 2. When a philosopher is thinking and wants to start eating, he has to execute a guarded action: the guard (stick x)$\wedge$(stick $((x+1) \bmod n)$) ensures that the needed sticks are both available, and the three (elementary) actions modify the theory by removing the two sticks, and by adding the eating philosopher.

2.3 Operational Semantics

Informally, the operational semantics of a component is defined via a transition system, the states of which are pairs consisting of a theory presentation, representing the current state of the store, and a process term, representing the current structure of processes to be run. The transitions between states are triggered by the execution of actions, either by processes, by the external environment, e.g., when a sensor has to be updated, or by the user via a dedicated interpreter for actions. At each state, the inference rules defining the operational semantics of the underlying declarative language may be run over the current store, e.g., one might ask for the philosophers which are eating. For a more detailed presentation of the operational semantics of our framework, the interested reader may consult [12].

3 User-Defined Actions

In the previous section, we have seen that actions are the principal constituents for the description of processes, since each run of a process corresponds to the performance of a possibly infinite sequence of actions. In this section, we discuss the definition of actions.

The actions executed by processes operate on stores. Notice that, whenever a process has to execute actions on some physical device, the latter is considered as a component, modeled as a store for the data description, together with processes for the control part of the device. This leads us to define an action as a curryfied recursive function which goes from stores to stores[3].

Definition 4. *An* action *possibly takes some arguments and returns a total recursive function from a store to a store, i.e., an action has the following profile*

```
action : arg_type_1 -> ... -> arg_type_n -> store -> store
```

where the sort `store` *denotes the ADT of stores, and the sorts* `arg_type_i` *denote the sorts of needed parameters.*

Using our framework, the integration of processes into a programming language $\mathcal{L}$ becomes straightforward as far as it is possible to define or to use actions which modify programs written in $\mathcal{L}$. Unfortunately, actions over programs are often supposed to be not a fundamental part of a language and are not specified for most familiar programming languages. Exceptions are *reflective* languages, as for instance Maude [6], or Common Lisp [28] together with its Metaobject Protocol [18]. In these languages, programs can be represented as data in the language itself, and these programs can be executed by an interpreter.

Nevertheless, even if classical programming languages do not provide actions as such, *particular* actions on programs exist, but are mixed with "predefined built-ins" of the syntax of the language. For instance, Prolog provides the "predicates" `assert` and `retract` which allow one to add and to remove clauses to and from a program. In SML (respectively, Scheme or Common Lisp), the "function" := (respectively, `setq`) permits to update the value associated to a "mutable cell". In Erlang, the "built-in functions" `load_module`, `delete_module` and `purge_module` allow to exchange a version of a module by a new, "corrected" one. These built-in actions, which are supposed to be used inside the sentences defining a program, have side-effects on that program and hence modify the program. Stated otherwise, the parameter representing the program to which the action is to be applied is missing, since it is implicitly "self".

There are many ways to define actions on programs for a given language $\mathcal{L}$, for example by providing a set of built-in actions or by providing an action description language (ADL). An ADL associated to $\mathcal{L}$ is defined as a language that allows the description of actions over programs of $\mathcal{L}$. Therefore, the structures of $\mathcal{L}$-programs have to be considered as data types of the ADL. Hereafter, we give hints for the definition of an ADL associated to a rule-based programming language as seen in Sect. 2. The ADL we use is a functional language, syntactically close to SML [22].

To let users define their own actions, stores should be represented as data objects of the ADL. Recall that a store of the language at hand is a triple $\langle S, \Omega, \mathcal{R}\rangle$ where S is a set of sorts, Ω is a family of sorted operations and $\mathcal{R}$ is a set of conditional rewrite rules. We give in Fig. 3 a description of the signature of an abstract data type (ADT) for the considered stores. This ADT is defined in a modular way. The signatures of basic data types such as strings and generic lists

[3] A similar view is taken by the authors of Concurrent Haskell (CH), when they call a *state transformer*, i.e., a function (or value) of type `IO t`, an *action* [24, section 2.1].

are missing. Data types of sorts, operations, variables, terms, rules and stores are described by their constructors (`make_?`), testers (`is_?`) and accessors (`get_?`). Note that this ADT is not meant to be an (optimized) implementation of the considered declarative language.

```
ADT store              (* STORES *)
make_store       : sort list -> operation list -> rule list -> store
get_sorts        : store -> sort list
get_operations   : store -> operation list
get_rules        : store -> rule list

ADT rule               (* RULES *)
make_rule     : term -> term -> term -> rule
get_lhs       : rule -> term
get_rhs       : rule -> term
get_condition : rule -> term

ADT term               (* TERMS *)
make_variable_term : variable               -> term
make_application   : operation -> term list -> term
is_variable        : term -> bool
is_application     : term -> bool
get_variable       : term -> variable
get_operation      : term -> operation
get_arguments      : term -> term list

ADT variable           (* VARIABLES *)
make_variable     : string -> sort -> variable
get_variable_name : variable -> string
get_variable_sort : variable -> sort

ADT operation          (* DEFINED OPERATIONS *)
make_operation     : string -> sort -> operation
get_operation_name : operation -> string
get_operation_sort : operation -> sort

ADT sort               (* SORTS *)
make_basic_sort      : string -> sort
make_functional_sort : sort -> sort -> sort
```

Fig. 3. Sample of the signature of an ADT `store`

4 Examples of Actions

In this section we give some examples of actions with their definitions. All actions are supposed to return well-formed stores.

4.1 Adding Rules

The actions which are most straightforward to define are those which just add something to a part of the store, for example the addition of a rule. In example 2, for instance, the (elementary) action (tell (is_eating x)) adds the rule "(is_eating x) $\rightarrow$ true".

Obviously, since in our example the store contains a *list* of rules, we have at least two different possibilities to implement this action, depending on the position where the new rule will be inserted into the list of rules. Two reasonable choices are for instance the beginning and the end of the list – the following is a (naive) specification of the former:

```
add_rule : rule -> store -> store
add_rule rule store =
  make_store (get_sorts store)
             (get_operations store)
             (cons rule (get_rules store))
```

Prolog provides two built-in "predicates", namely `asserta` and `assertz`, which add a new clause at the beginning (`asserta`) or the end (`assertz`) of the clauses defining the corresponding predicate [9, pages 44 – 47].

Obviously, the possibilities will be different, if the ADT of stores is more sophisticated in order to implement "optimal" evaluation strategies, where rules are stored, for instance, within definitional trees [2].

There are still more possibilities for the addition of a rule to a store. Consider the case of adding several times the same rule. Depending on the operational semantics, the existence of duplicated rules may be important (notice that the standard of Prolog [9] is not very precise about how this is handled in Prolog). In classical ccp for instance, telling the constraint c several times (i.e., adding the constraint c several times to the store) is equivalent to telling it just once [26].

The addition of a rule could also be combined with a test, such as adding a rule only if it is not yet present in the store (modulo an equivalence relation).

4.2 Removing Rules

There are several possibilities to remove a rule from a store. A programmer might want to remove a precise rule, or all rules of a specified form. Thus, when removing a rule, we should test each rule separately if it should be removed or not. Different possibilities of removal will then correspond to different `test`s, which might be a functional parameter of the action. Examples for such `test`s are identity, identity up to renaming of free variables, pattern matching, unification, equality (equivalence with respect to the store), etc. A possible implementation might be the following:

```
remove_rules : (rule -> bool) -> store -> store
remove_rules test store =
   make_store (get_sorts store)
              (get_operations store)
              (find_all (fun x -> not (test x)) rules)
```

where the function `(find_all t list)` returns the list of all elements `e` of the list `list` for which the evaluation of `(t e)` returns true. We use an anonymous function (or λ-abstraction) to inverse the result of the test `test`, that is to say the expression `(find_all (fun x -> not (test x)) rules)` denotes the list of all rules `r` in the list of rules `rules` for which `(test r)` returns `false`.

In Prolog, the built-in "predicate" `abolish(predicate)` removes all clauses for a given predicate `predicate` (in a single step), whereas `retract(pattern)` removes all rules which can be unified with the rule-pattern `pattern` (one by one upon backtracking) [9, pages 37 & 38, 154 & 155]. While `abolish` is not very "precise", the successful use of `retract` requires to control the number of necessary backtracking steps, which is to our opinion not straightforward.

4.3 Assignment

Probably the most common action is assignment (:=) as it is ubiquitous in imperative programming languages and also used in some declarative languages as SML [22], Common Lisp [28] or Oz [27].

Similar to the view of assignment in the coordination language Linda [14, page 98], we see the assignment `c := term` as removal of all rules defining the (constant) operation `c` and addition of a single rule which redefines `c` to have the value `term`. Thus, using the actions defined above (i.e., the functions `add_rule` and `remove_rules`), the action of assignment might be defined as follows:

```
(:=) : operation -> term -> store -> store
(:=) operation term store =
(add_rule
   (make_rule (make_application operation nil) term true)
   (remove_rules
      (rule_pattern_match
         (make_rule (make_application operation nil)
                    (make_variable_term x)
                    (make_variable_term y)))
      store))
```

where `x` and `y` are variables, `true` is the boolean constant true, and `nil` the empty list. `rule_pattern_match` is a test function which implements a pattern-matching-based removal[4]:

```
rule_pattern_match : rule -> rule -> bool
rule_pattern_match rule1 rule2 =
   (term_matches (get_lhs rule1) (get_lhs rule2)) and
   (term_matches (get_rhs rule1) (get_rhs rule2)) and
   (term_matches (get_condition rule1) (get_condition rule2))
```

where we use the standard pattern matching function `term_matches`:

```
term_matches : term -> term -> bool
```

[4] We suppose that `and` is evaluated "lazily" in a sequential manner from left to right.

`(term_matches term1 term2)` returns `true` if `term1` matches `term2`.

Note, that in our framework, a "variable" in the sense of standard imperative programming languages corresponds to a "changing constant": using assignment, we may change the theory which defines the value of the constant, and in each of these theories, the value does not change, i.e., it is constant.

A further possibility for the assignment action (`c := term`) might reduce the term to normal-form, i.e., the action might add a rule defining `c` to have the value `term'` where `term'` is the normal-form of `term`.

4.4 Modifying the Signature

So far, we have only considered the modifications of the rules of a store. Modifications of the other part of the store, i.e., its signature, might be interesting too. For instance, consider an implementation of a window system. Such a system needs to store information about all the different windows that are currently displayed on the screen. Roughly speaking, a theory describing the current state of the system might model every window by a constant. Hence, when a request for the creation of a new window arrives, the theory has to be changed, and a *new* constant corresponding to the new window needs to be created. A similar example is the dynamic creation of new communication channels, which is mandatory in order to model mobility through link passing as in the π-calculus [21]. These examples are presented in more detail in [12].

These examples have in common that the enrichment of the signature is limited to new constants, which then may be further used and modified by assignment as in classical imperative programming languages. However, there are also situations where the addition of a new operation, or even a new sort might be necessary. For instance, if a program has to be modified, the new version of the program might use new operations over new data-structures, in particular new data-types. This happens for example if we want to change the implementation of an algorithm using another, more efficient data structure, as for instance graphs instead of lists.

Actions modifying the signature are executed implicitly in some of the interactive interpreters for modern declarative languages, whenever the definition of new global symbols is permitted, as for example the `let`-construct in `ocaml` and SML/NJ.

Removing declarations from the signature could be more problematic. For example removing an operation from the signature should also remove all the sentences (rules) including that operation.

4.5 Complex Actions

More complex actions are needed when particular parts of a store have to be modified. A typical example is the correction of errors without stopping the activity of the entire system, as it is required for example for large telephone exchanges or air traffic control systems.

To handle such systems, the concurrent functional language Erlang [3] provides the built-in functions `delete_module`, `load_module` and `purge_module` which allow to replace a complete module by a new, updated version (without stopping the entire system). Since Erlang is untyped, this allows also the modification of the signature of the module, by changing the profile of functions, and the set of symbols defined in the module. But one might imagine situations where the exchange of an entire module is too coarse-grained, and where the modification of a single rule is sufficient.

Maybe we need even to change the profile of some functions. Consider a function computing the price of train tickets. There may be new, unpredictable policies, such that the age of the customer, the season or the time-period has to be taken into account. Thus the function computing the prices of train tickets has to be replaced (in the entire store) by a new one, which has some additional parameters. This action (in its generality) is difficult (or even impossible) to express using the built-in actions of existing declarative languages. Another example are chip-cards. Since about two years are needed for the design of the cards, the requirements of the application have probably evolved. Hence, it would be interesting to have the possibility to adapt the program in chip cards without the need for redesigning the chip.

5 Related Work

As already mentioned in the preceding sections, most existing programming languages do not provide actions for the modification of programs as such, but rather as special built-in primitives integrated into the syntax of the language. These primitives are usually difficult to adapt to languages based on a different paradigm.

Since examples for such primitives are too numerous to survey them all, we restrict ourselves to give just a selection of some different kinds. When they have already been discussed in Sect. 4, we do not further comment them here. For instance, the logic language Prolog [9] provides the built-in predicates `assert`, `retract` and `abolish` which allow one to add (respectively, remove) clauses to (respectively, from) a program (see Sects. 4.1 and 4.2). In languages based on the framework of `ccp` [26], as for instance Oz [27], the action tell allows to add a constraint to the global constraint store (see Sect. 4.1). Languages providing assignment are numerous. Besides imperative programming languages (e.g., C++, Java, ADA, etc.), assignment actions can be found in functional languages (e.g., SML or Common Lisp) or constraint programming languages (e.g., Oz) (see Sect. 4.3). The concurrent functional language Erlang [3] offers the built-in functions `delete_module`, `load_module` and `purge_module` which allow users to replace a module by a new corrected one without stopping the entire application (see Sect. 4.5). CML [25], Erlang [3] and Concurrent Haskell (CH) [24] provide a built-in primitive `spawn` (or `forkIO` in CH) the "side-effect" of which is to launch a concurrent process. Interprocess communication via ports (e.g., in AKL [16, 17], Oz [27] and Curry [7, 15]) uses the built-ins `openPort` and `sendPort` for the creation and the sending of messages through ports. The latter primitive modifies the program by adding the message *at the end* of the associated stream; thus the information about the current end of the stream has to be kept in the program and to be modified when sending a message.

Similar to most concurrent declarative programming languages, most approaches to state changes in (logic) databases [5] do not clearly distinguish between the notions of actions and predicates. Also, by focusing on database updates which can be rolled back, these approaches do not encompass all actions, in particular actions that manipulate physical objects external to the system, as for instance alarm bells or production machines, since one cannot "roll back" the emitted sound or reverse every mechanical transformation.

Additionally, all these built-ins are required to operate on the program executing them. In other words, all actions are implicitly applied to the store itself, as for example the implicit state parameter of the monadic I/O in CH [24]. Consequently, these languages are in fact designed for the description of self-contained components and thus, additional tools are necessary for the coordination and communication of components written in such languages. An example of such a tool are *coordination* languages, e.g., Linda [14]. In Linda, processes communicate via a shared tuple space, using several primitive actions, namely primitives for adding, reading and removing tuples. To allow the definition of more high-level communication schemes, the idea of programmable coordination media has been suggested [8]. Similar to our approach, a programmer can define the actions executable on the tuple space. By using (declarative) programs instead of a tuple space, we give the programmer the possibility to express more complex coordination structures on an abstract level, without the need to *encode* these strategies via tuple space operations.

Reflective programming languages, e.g., Maude [6] or Common Lisp [28] together with its Metaobject Protocol [18], allow the representation of programs as terms of the language itself. Therefore, the language itself can be used as an ADL for the definition of actions, since the language corresponds to its own meta-language. In Maude, the module `META-LEVEL` allows the modification of the modules which are stored in a "module-database". Metaobject Protocols (MOP's) [18], as for instance in the object system of Common Lisp [28], provide also the possibility for the definition of actions on parts of the program. In JAVA [20], reflection is limited to the discovery information about objects of the program, to the invocation methods (including constructors for the creation of new objects) and to the modification of fields. However, no new classes or methods can be defined.

In combination with an implementation of an interpreter of the language (in the language), as for example the special operator `eval` of Common Lisp, which interprets its argument, reflection allows to write self-modifying programs. This can be used to implement programs the execution scheme of which is similar to our components. However, since the definitions of processes is not separated from the store, even the definition of processes can also be modified dynamically.

To facilitate the understanding of programs, we prefer to separate the different matters, namely the store and the processes modifying the store.

The scripting language Tcl/Tk [23] allows to modify stores representing graphical user interfaces (GUI's) by the execution of commands. The set of available actions (or commands) can be extended. Therefore, thanks to its adaptability and flexibility, Tcl/Tk is widely used for building GUI's.

6 Conclusion

In this paper we have presented the definition of actions in a framework for concurrent declarative programming. In this framework, actions are defined as functions from programs to themselves, using a metalanguage where abstract data types of programs are available. Most existing (declarative) languages provide such actions as built-in primitives which are presented as a part of the syntax, whenever they are provided at all. Hence the integration of these languages into our framework is unfortunately not straightforward.

In fact, to be integrated into our framework, a programming language should provide the possibility to define the actions on stores, for instance by providing an action definition language (ADL), where programs are considered as data objects. A natural way to provide such an ADL is to include an ADT of programs in the language definition, in addition to the syntax and semantics.

We would have appreciated to have such ADT's for existing languages, as we could have integrated them easily into our prototype of a multi-paradigm platform currently under development.

References

1. J.-R. Abrial. In *Formal Methods for Industrial Applications*, vol. 1165 of *LNCS*, chap. Steam-Boiler Control Specification Problem, pp. 500–510. Springer, 1996.
2. S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proc. of ALP 1992*, vol. 632 of *LNCS*, pp. 143–157, Sept. 1992. Springer.
3. J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in ERLANG.* Prentice Hall, 1993.
4. J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.
5. A. J. Bonner and M. Kifer. The state of change: A survey. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases: Invited Surveys and Selected Papers of DYNAMICS '97*, vol. 1472 of *LNCS*, pp. 1–36, 1998. Springer.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic.* Computer Science Laboratory, SRI Int., Mar. 1999.
7. Curry: An integrated functional logic language. available at `http://www.informatik.uni-kiel.de/~mh/curry/report.html`.
8. E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In D. Garlan and D. L. Metayer, editors, *Proc. of Coordination '97*, vol. 1282 of *LNCS*, pp. 274–288, Sept. 1997. Springer.

9. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard, Reference Manual*. Springer, 1996.
10. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chap. 6, pp. 243–320. Elsevier, Amsterdam, 1990.
11. E. W. Dijkstra. Hierarchical ordering of sequential processes. In C. A. R. Hoare and R. H. Perrott, editors, *Proc. of a Seminar on Operating Systems Techniques*, vol. 9 of *A.P.I.C. Studies in Data Processing*, pp. 72–93, Belfast, 1971. Acdemic Press.
12. R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In J. Lloyd et al., editors, *Proc. of CL 2000*, vol. 1861 of *LNAI*, pp. 300–314, London, July 2000. Springer.
13. R. Echahed and W. Serwe. A component-based approach to concurrent declarative programming. In *Proc. of WFLP 2001*, Kiel, Sept. 2001. This volume.
14. D. Gelernter. Generative communication in Linda. *ACM TOPLAS*, 7(1):80–112, Jan. 1985.
15. M. Hanus. Distributed programming in a multi-paradigm declarative language. In G. Nadathur, editor, *Proc. of PPDP '99*, vol. 1702 of *LNCS*, pp. 188–205, 1999. Springer.
16. S. Janson. *AKL–A Multiparadigm Programming Language*. PhD thesis, Uppsala Theses in Computing Science 19, June 1994. SICS Dissertation Series 14.
17. S. Janson, J. Montelius, and S. Haridi. Ports for objects in concurrent logic programs. In G. A. Agha, P. Wegner, and Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.
18. G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*, chap. 5 and 6. The MIT Press, 1991.
19. J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pp. 1–112. Oxford University Press, 1992.
20. G. McCluskey. Using Java reflection. Jan. 1998, available at `http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection`
21. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.
22. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML–Revised*. The MIT Press, 1997.
23. J. K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, 1994.
24. S. L. Peyton Jones, A. D. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL '96*, pp. 295–308, Jan. 1996.
25. J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
26. V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.
27. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, vol. 1000 of *LNCS*, pp. 324–343. Springer, 1995.
28. G. L. Steele, Jr. *Common Lisp the Language*. Digital Press, second edition, 1990.
29. P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315–351, Feb. 1998.