Prototyping an Inconsistency Checking Tool for Software Process Models

Jin-Kao Hao, Jean-Jacques Chabrier, Francois Trousset

Prototyping an Inconsistency Checking Tool for Software Process Models

Jin-Kao Hao[1], François Trousset[1], and Jean-Jacques Chabrier[2]

(1) LERI/EERIE
Parc Scientifique Georges Besse
F-30000 Nîmes
France
email: hao@eerie.eerie.fr
fax: (33) 66840506
phone: (33) 66387000

(2) Centre de Recherche en Informaique de Dijon
Faculté des Sciences Mirande
B.P. 138
F-21004 Dijon Cédex
France
fax: (33) 80395069
phone: (33) 80395881

Abstract: Software process modeling has attracted much research effort in Software Engineering. However, there is little work reported for the verification of process models. In fact, the verification is often either performed by hand or it is left to the enacting mechanism to detect inconsistencies during execution. Since process models are becoming more and more powerful and complex, their verification is also becoming increasingly difficult and critical. Our proposition is that in the same way as we need process modeling to facilitate software manufacturing, we need special tools to help verify the consistency of software process models. This paper presents part of our research and our experience in designing and prototyping such a tool for the verification of software process models in the ALF project (Esprit No.1520). The tool helps verify the partial consistency of process models by statically detecting various inconsistencies. This prototype uses techniques developed in different fields such as compilation, constraint solving and logic. To our knowledge, this is the first tool of its kind designed for the static checking for process models.

Keywords: Software process modeling, inconsistency checking, Prolog application

---

* This work was partially supported by the ALF Project (Esprit No. 1520). The work was completed when the first two authors were at the Centre de Recherche en Informatique de Dijon.

1. Introduction

Recently, there has been growing interest in the software process: the process for software manufacturing. Taking a view originating from the software process is an important advance in Software Engineering. According to Mark Dowson, "we are part of an industry that manufactures (and maintains and evolves) large, complex artifacts -- software systems. We would like our products to be of high quality and reasonable cost and to be delivered on time and within budget. The only way to achieve these objectives is to focus on the manufacturing process -- the software process" [Dowson 91]. Informally, a software process is a set of activities for producing and evolving software systems throughout the software life cycle.

Associated with the process view is process modeling. A process model is a static description which specifies the general features of a class of software processes, but not those features specific to a particular process. In other words, a process model governs each process of this modeled class.

A process model can thus be tailored to different processes by instantiating the process model entities with specific tools, objects, and other resources. The relationship between a process model and processes modeled by the model can be considered as an inheritance relation. In fact, suppose that we have a Software Process Model, SPM, which models the Software Processes, SP1, ..., SPk. Suppose also that SPM possesses certain properties, then each process SPi derived from SPM will inherit these properties, specified in the initial process model. The process modeling activity, like the program writing activity, is not free of error. Thus, as properties are inherited from the model to software processes, it becomes evident that it will be of great benefit to detect and eliminate as many of those errors as possible (called inconsistencies in this paper) in a given process model before giving it to an enacting mechanism for execution or interpretation. The *static* detection of inconsistencies is part of what we call the *verification of software process models*.

Although process modeling has attracted much research effort, there is relatively little reported work for the verification of process models. In fact, the verification is often either performed by hand or it is left to the enacting mechanism to detect inconsistencies during execution. Since process models are becoming more and more powerful and complex, their verification is also becoming increasingly difficult and critical. Our proposition is that in the same way as we need process modeling to facilitate software manufacturing, we need special tools to help verify the consistency of software process models. This paper reports part of our research and our experience in designing and prototyping such a tool for the *static* verification of software process models in the ALF project (Esprit No.1520). The tool helps verify the partial consistency of process models by statically detecting various inconsistencies. This prototype uses techniques developed in different fields such as compilation, constraint solving and logic. It especially uses "general type solving techniques" developed in [Trousset 92] to deal with specific typing

problems in process modeling. Additionally, our experience shows, once again, that Prolog is a very good candidate for rapid and evolving prototyping in some research areas.

The presentation is organized as follows: section 2 briefly reviews the MASP concepts, the process model proposed in the ALF project as well as the MASP/DL, the description formalism of MASPs; in section 3, we present the strategy used for prototyping our checking tool as well as the technical choices such as the prototyping environment and language; section 4 gives an overview of the tool itself and the techniques used in the tool; section 5 discusses future work; the last section concludes the paper with some observations and perspectives.

2. Software Process Modeling in the ALF Project

As with any programming language where a compiler depends on the syntax and semantics of the language, a verification tool for software process models depends on the syntax and semantics of the description formalism used for process modeling. In order to present our inconsistency tracker, we will give a brief reminder of the MASP concept, the software process modeling in the ALF project and the MASP/DL, the associated description language.

2.1 Process Modeling - the MASP Concepts

In ALF, software processes are modeled in terms of MASPs [Benali et al. 89, Griffiths et al. 89]. MASP stands for Model for Assisted Software Process. A MASP M is composed of six elements <OM, OP, EX, RU, OR, CH> (some elements are optional) where

OM - Object Model specifies the object types involved in the MASP as well as the relationship between those object types. The object model is the PCTE object model [Bourdier et al. 88] which is based on the Entity Relation model.

OP - OPerator types specify classes of operators. An operator type is defined by a name, a signature, and a pre- and a post-condition. The signature specifies the types of input/output objects manipulated by the operator type. The pre- and the post-conditions specify respectively constraints which must be satisfied before and after activating an operator of the operator type. An operator type may be instantiated by an existing tool (e.g. a C compiler) or a tool defined itself by a MASP which specifies a process submodel. This second kind of operator description introduces the notion of MASP-structure (MASP-hierarchy).

EX - Each EXpression definition consists of three parts: a name, an optional event and a logic expression. By naming expressions, it is possible to refer to them by their names without rewriting them. This is similar to macro definition and expansion. The event part of an expression defines the moment to evaluate the

logic expression part. Expressions are used as part of pre- and post-condition of operator types, characteristics, and the condition part of rules (see below).
The expression model is optional.

RU - RUles describe under which conditions execution of an operator may be attempted.

A rule contains two components. The first is an expression, the second is the name of an operator. The expression part of the rule and the pre-condition part of the concerned operator together constitute the necessary condition of the operator's execution. The rule model is optional.

OR - ORderings specify constraints on the order in which operators may be executed. Orderings are used to specify that certain operators can be executed concurrently, sequentially, optionally, simultaneously or repeatedly. This part is optional.

CH - CHaracteristics are expressions which should be true in each state of a software process. Characteristics can be used to describe integrity constraints and software process goals. This part is optional.

2.2 Process Modeling Formalism - the MASP Definition Language

Generally speaking, we can distinguish two ways of representing software process models. First, process models can be described with the aid of an informal or narrative formalism such as natural languages and flowcharts, which has been long employed by many organizations. Second, they can be described by means of a formal notation such as programming language-like formalisms, which is exemplified by the *process programming* [Osterweil 87]. The discussion about the advantages and drawbacks of the informal and formal approaches can be found elsewhere; for example, see [Keller 88, Lehman 87, Osterweil 87].

The process modeling in the ALF project is based on the second approach. A formal language with well defined syntax and semantics was developed. This is the MASP Definition Language (MASP/DL for short). MASP/DL is a powerful, yet very complex formal modeling language. Given that there is no space to present the entire language, here, we only give an example of a MASP written in MASP/DL. This will help the reader to grasp the outline of the language.

```
MASP Example_1 HAS_TYPE program_development;
OBJECT_MODEL_IS
      NEW_SDS example_sds IS
            IMPORT sys-object AS object; IMPORT sys_file AS file;
            ready_for_compilation : BOOLEAN;

            e_file       : SUBTYPE OF file;
            c_module     : SUBTYPE OF e_file;
            obj_module   : SUBTYPE OF file;
            err_file     : SUBTYPE OF file;

            error        : COMPOSITION LINK TO err_file;
            obj          : COMPOSITION LINK TO obj_module;
```

```
            EXTEND e_file WITH ATTRIBUTE ready_for_compilation END;
      END example_sds
END_OBJECT_MODEL;


OPERATOR_MODEL_IS
      edit : (INOUT _file : e_file)
            PRECOND      : TRUE
            POSTCOND     : ready_for_compilation(_file, TRUE)
            KIND         : INTERACTIVE
      compile : (IN _m : c_module; OUT _o: obj_module; OUT _err: e_file)
            PRECOND      : ready_for_compilation(_m, TRUE)
            POSTCOND     : obj(_m,_o,TRUE)
            KIND         : NON_INTERACTIVE
      display_err : (IN _m : c_module)
            PRECOND      : error(_m, _err, TRUE)
            POSTCOND     : -
            KIND         : NON_INTERACTIVE
      exit : ()
            PRECOND      : TRUE
            POSTCOND     : -
            KIND         : INTERACTIVE
END_OPERATOR_MODEL;

RULE_MODEL_IS
    IF IT_EXISTS _m: c_module SUCH_THAT NOT obj(_m,_obj,TRUE)
            THEN compile(_m, _obj, _err);
    IF FOR_ALL _m: c_module IT_EXISTS _o: obj_module SUCH_THAT obj(_m,_o,TRUE)
            THEN exit();
    IF IT_EXISTS _m: c_module AND _e: err_module SUCH_THAT error(_m,_e,TRUE)
            THEN edit(_m);
END_RULE_MODEL;

ORDERING_MODEL_IS
      order: FOR_ALL _m: c_module DO [edit(_m)]; compile(_m,_o,_err);
                                          (* display_err(_m) *)
END_ORDERING_MODEL;
END_MASP;
```

An Example of MASP Written in MASP/DL

This MASP models a simplified "edit<->compile" development cycle and is by no means realistic. It should be noted that MASPs developed in the ALF project are far more complicated, and always organized into a hierarchy. However, this MASP does highlight some important modeling aspects with the MASP/DL. All object types are declared in the object model; they are all descendants of a predefined object type "sys-object". Object types are structured into a hierarchy. Four operator types are defined. Each operator is composed of a precondition, the operator type (which may be a basic tool or another MASP) and the types of objects involved, a postcondition and the mode of the operator type (the *interactive* mode indicates that the operator can be invoked by the user). The rule model represents part of the process knowledge. For example, the third rule says that we terminate the process if all `c_module` are successfully compiled. The ordering specifies that an object `_m` of type `c_module` can be edited any number of times (though at least once) before being compiled and then the compilation errors are displayed if necessary.

2.3 Inconsistencies in MASPs

Evidently, inconsistencies (including syntax errors) may be introduced when writing MASPs. This situation is very similar to that of writing programs in any programming language. However the problem here is much more difficult due to the complexity of the MASP/DL. Moreover, we must verify not only the syntax and semantics of the MASP/DL, but also part of the semantics of the models specified by the MASPs.

Inconsistencies in a MASP can be roughly classified into three categories, called A, B and C respectively (syntax errors are not classified here).

A Class: Errors concerning declaration/references. The following examples show this kind of error.

A rule refers to an operator name which is not defined in the operator model, or the arity of the reference does not fit its definition. Any reference to an undefined entity (object, attribute, link, expression, ...) is also an error.

A parameter of an operator type has double definition.

An entity (object, attribute, link, expression, ...) has double definition.

and so on.

B Class: Typing problems. Some typing problems in MASPs are similar to that of any programming language. However, due to the complexity of the MASP/DL, typing problems here are more general and difficult than those found in traditional languages. Consequently, the checking and inferencing methods developed in the compiler construction for type checking are in general not sufficient. We will come back to the discussion later. Now let us look at an example.

```
MASP Example_2 HAS_TYPE program_development;
OBJECT_MODEL_IS
      NEW_SDS example_sds IS
            IMPORT sys-object AS object; IMPORT sys_file AS file;
            incl          : SUBTYPE OF file;
            generic_type  : SUBTYPE OF file;
            module        : SUBTYPE OF object;
            l1_module     : SUBTYPE OF module;
            l2_module     : SUBTYPE OF module;
            l3_module     : SUBTYPE OF module;
            application   : SUBTYPE OF object;

            has_module    : COMPOSITION LINK TO l1_module, l2_module;
            has_include   : COMPOSITION LINK TO incl;
            has_generic   : COMPOSITION LINK TO generic_type;

            EXTEND l1_module WITH ATTRIBUTE has_generic END;
            EXTEND l2_module WITH ATTRIBUTE has_include END;
            EXTEND l3_module WITH ATTRIBUTE has_include, has_generic END;
            EXTEND application WITH ATTRIBUTE has_module END;
      END example_sds
END_OBJECT_MODEL;

OPERATOR_MODEL_IS
      compile : (IN _a : application)
            PRECOND       :     has_module(_a, _m, NO_KEY)
                          AND   has_include(_m, _i, NO_KEY)
                          AND   has_generic(_m, _g, NO_KEY)
            POSTCOND      : -
            KIND          : NON_INTERACTIVE
END_OPERATOR_MODEL;
END_MASP;
```

This MASP describes a fictitious model for applications development in which an application is composed of modules of three different languages L1, L2 and L3. L2 modules can have inclusion files (`incl`), those of L3 both inclusion files and generic types (`generic_type`), and those of L1 generic types. An operator type (`compile`) is defined to allow compiling applications. The precondition of the operator type is an expression which defines the kind of applications which can be compiled. As will be shown in section 4, the expression cannot be typed correctly since there is no possible type for the variable `_m` satisfying both the declarations for the objects and the precondition of the operator type.

***C Class:** This kind of inconsistency can be expressed as a contradiction in a logic expression or in a conjunction of logic expressions. In order to simplify our presentation, we introduce some notations.

Let **OP** be an operator, then **Pre(Op)** and **Post(Op)** represent respectively the logic expression part of the pre- and post-condition of **Op**. Let **R** be a rule, then **Cond(R)** is the logic expression of the condition part of the rule **R** and **Oper(R)** the operator used in R. We also use **Charact(M)** to designate the logic expression of the characteristic of the MASP **M**. With these notations, logical inconsistencies can be defined at two levels.

1) in an expression which is a **Pre(Op)** or **Post(Op)**, a **Cond(R)** or still a **Charact(M)**. For example, if **Pre(Op)** is always false, then the precondition **Pre(Op)** of the operator **Op** can never be satisfied. This inconsistency means that **Op** can never be executed. Similar inconsistencies may occur in **Post(Op)**, **Cond(R)** or **Charact(M)**.

2) in a conjunction of logic expressions which are used in different components of a MASP. For example, inconsistencies may occur between an operator and a rule. More precisely, if the conjunction **Pre(Oper(R))** and **Cond(R)** is always false, then there is an inconsistency because the rule can never be used. Similar inconsistencies may occur for the interaction between operators, rules, orderings and characteristics.

The presentation above gives only a partial view of the different inconsistencies in MASPs, other inconsistencies are also possible. For example, cycles may exist in rules, in an object model due to subtyping etc. See [Chabrier et al. 89] for more detail.

Having presented the problems with which our Inconsistency Tracker For MASPs (ITFM for short) is confronted, we are now ready to describe our prototyping strategy for the design of the tool itself.

3. Prototyping Strategy

A prototype is a simplified model and can be built rapidly and modified easily. However, a prototype is itself a piece of software, even a complex system. This implies that a prototype cannot be built in an "ad hoc" way. In order to be productive, a prototype should be built with

great care. It needs good preparation, and good organization. It also needs appropriate techniques and tools. In other words, it needs a good *prototyping strategy*.

The prototyping strategy used is similar to that of [Penedo 86]. The following steps have been established and followed for the prototype development:

• Identify research issues. Given the variety of inconsistencies that may be encountered in MASPs, we have identified the following research issues for investigation during the prototyping excises.

* Internal representation of MASPs. MASPs are written in MASP/DL. MASP texts may be very complex. How to represent MASPs internally constitutes the first question to be answered. An appropriate internal representation is needed in order for the ITFM to track various inconsistencies, both syntactic and semantic. A good representation will facilitate the implementation of the prototype while a bad one complicates the task.

* Type solving (A & B Class inconsistencies). As we have seen above, type inconsistencies in MASPs may be difficult to track. The purpose of the type-checkers here is not really to synthesize the type of each expression from the types of its subexpression. Instead we want to deduce, from declarations, the possible types of each variable (a variable represents an undeclared identifier or a subexpression, and may have several possible types) in an expression, and if needed, the types for each subexpression can be synthesized from the types of each variable. We use the term "type-solving" to distinguish this difference with traditional type-checking problem. We need more general, more powerful methods to deal with typing problems in MASPs.

* Logical level inconsistency checking (C Class inconsistencies). The fact that a MASP involves only types (not concrete objects) implies that the ITFM must deal with general logic expressions. Also, the MASP/DL allows the using of extra logic structures (predifined predicates of the MASP/DL) inside expressions we must deal with. Finally, an expression may contain non quantified variables (called free variables). Special techniques are then needed to track logic inconsistencies.

• Propose possible solutions. For each identified research issue, we have studied possible solutions.

* The internal representation of MASPs is dependent on the prototyping language and environment. Following our experience with Prolog, we have decided to represent MASPs as Prolog facts in Prolog's database (see §4.1). If we had chosen another language, other representations, e.g. a database would have been used to store MASPs.

* Type solving. The essential point of a type system is to decide whether each variable of an expression can be typed correctly. This problem can be restated in another way; i.e. we want to know if each variable can have at least one possible type while satisfying all the concerned declarations. This statement is very similar to the definition of the Finite Constraint Satisfaction Problems [Hao & Chabrier 90, 91, Chabrier et al. 91]. Indeed, this problem can be dealt with

using constraint solving techniques. Techniques for general typing problems developed in [Trousset 92] are considered applicable. This will be discussed in the next section.

* Logic inconsistency checking. As expressions are first order predicates, methods in logic programming can be applied. However, due to the impure features (predefined predicates) and the generality of the expressions in MASPs, the direct application of predicate logic will lead to superfluous processing. This problem is solved by using rewriting techniques to express the predefined semantics of these predicates.

• **Choose prototyping environments.** The development environment and the languages used may have a great influence on the success of prototyping activities. Our experiences with Prolog as well as that of others, has convinced us of its suitability for prototyping. Sepia Prolog [Meier et al 88] is chosen as our basic development language. A workstation under Unix constitutes the working environment.

• **Build and experiment with the prototype.**

• **Identify further work.**

4. An Overview of the ITFM Tool

This section gives an overview of the ITFM prototype, its organization and the techniques used in the prototype.

The ITFM is essentially composed of four independent modules: a parser, two type-checkers, and a contradiction checker. Figure 1 illustrates the organization of the components of the prototype.

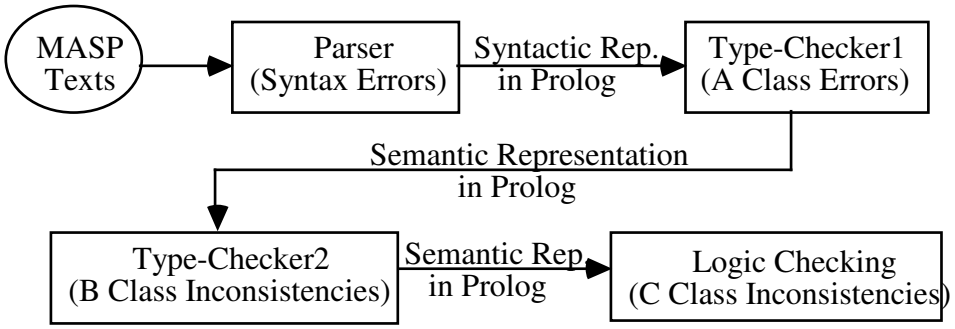

Fig. 1 Organization of the ITFM

4.1 Parser

The role of the parser is to take a MASP as its input, parse it and produce an internal representation in Prolog of the given MASP. In this representation, each component of the MASP is coded into several Prolog facts. For example, given the definition of an operator type in a MASP:

```
compile : (IN _m : c_module; OUT _o: Obj_module; OUT _err: e_file)
       PRECOND      : ready_for_compilation(_m, TRUE)
       POSTCOND     : o(_m,_o,NO_KEY)
       KIND         : NON_INTERACTIVE
```

it will be translated into the following set of Prolog facts:

```
oper_type(21, operator_2, 'compile, non-interactive).
oper_args(21, operator_2, 1, '_m', name('c_module'), in).
oper_args(21, operator_2, 2, '_o', name('obj_module'), out).
oper_args(21, operator_2, 3, '_err', name('e_file'), out).
oper_prec(22, operator_2, 3,
       eval('ready_for_compilation',[name('_m'), 'TRUE'])).
oper_post(23, operator_2, 3,
       eval('o', [name('_m'), name('_o'), [])).
```

In this representation, every syntactic detail of the definition is recorded. Besides, other information such as the line number and position number of each parameter of the operator type is added in order to convey diagnostics in case of the detection of errors.

The parser essentially uses the syntax-directed techniques developed in compilation [Aho et al. 86]. It works with the formal definition of the MASP/DL grammar. During the parsing phase, syntax errors are reported whenever detected. However, the ITFM stops after reporting a syntax error. This is reasonable since the ALF project supposes that MASPs are written with the MASP/Editor that guarantees the syntactic correctness of MASP texts. There are two interesting points to note. First, instead of producing a parse tree as in a compiler, our parser produces Prolog facts which represent exactly the relevant syntactic information of a given MASP (irrelevant information, for example ';' is eliminated). Second, the choice of Prolog as our prototyping language makes this phase simple, it should be noted that Prolog was initially invented for natural language processing.

4.2 Type-Solving

Type-checker 1. Its role is to track most of the inconsistencies defined in the A class, i.e. errors concerning declarations/references. The checker consists of a set of checking rules specifying the condition of each kind of inconsistency. Thanks to the internal representation of MASPs and the deduction mechanism of Prolog, the writing of checking rules is easy. For example, the checking rule for duplicated operator definition says that we should first collect all the operator names of a MASP in a list, and then verify if there are two identical names in the list. The type-checker 1 directly uses the syntactic information represented by the internal representation. During the checking, some semantic information is deduced and stored in the Prolog database. At the end of the type-checker 1, the first internal representation is transformed into the second representation (always in the form of Prolog facts) replacing some syntactic information (e.g. the names) with some semantic information (e.g. a link to their definition). This representation constitutes the starting point of the type-checker 2 which is based on constraint solving methods.

Type-checker 2. Its role is to track B class inconsistencies. That is, given an expression, it checks if each variable of the expression can have at least one possible type which satisfies all the related declarations. Such a typing problem can be considered as a finite constraint satisfaction problem CSP(V,D,C) (since the set of types defined is finite) where

V={V1, V2, ..., Vn}, a set of variables.

D={D1, D2, ..., Dn}, a collection of sets of possible values for each variable of V,
Di called domain of Vi.

C={C(Vi1, Vi2, ..., Vij) | C is a relation on Vi1, Vi2, ..., Vij of V}.

Given a Typing Problem TP, we first need to transform the problem into an equivalent CSP. Then solving the CSP means solving the initial TP. This is presented in figure 2.

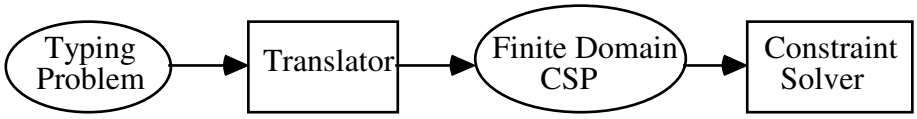

Fig. 2 : Structure of the type-checker 2

In figure 2, the constraint solver poses no problem, since we know that well studied constraint solving techniques exist. Once the TP is transformed into a CSP, we know how to solve it by using constraint solving techniques. So the typing problem now involves making the transformation. For a given TP, there are many ways to transform it into a CSP. However, whatever transformation is used, the resulted CSP will be equivalent if the transformation used is sound and correct.

The transformation prototyped in the type-checker 2 is now briefly presented using a simple example. The main idea is to associate a solver variable for each expression variable, each subexpression and to consider each expression to be checked as a set of constraints on these variables, each variable having all possible types as its domain at the beginning.

Let us look again at the MASP example_2 given in section 2. We want to check the expression of the precondition of the operator `compile`. At the beginning, the variable `_a` already has a specific type `application`, while `_m, _i, _g` can be any type defined in the object model. The semantic of the predicate "`has_module(_a, _m, NO_KEY)`" says there is a link called "has_module" from `_a` (of type application) to `_m` (of unknown type). First, we check if such a link exists. The answer is yes because in the object model the type `application` is effectively extended with a link "has_module". Next, from the definition for the links, we know that the destination of a link of type `has_module` is `a_module` or `c_module`. Therefore the type of `_m` belongs to {`a_module, c_module`}. The second predicate of the expression "`has_include(_m, _i, NO_KEY)`" says there are links named "has_include" from `_m` to `_i`. Checking the object model, we know the object `c_module` or `p_module` have been extended

with such a link. We thus deduce that the type of _m belongs to {c_module, p_module}. In the same way, we deduce, from the third predicate "has_generic(_m, _g, NO_KEY)" that the type of _m must belong to {a_module, p_module}. Since the three predicates are connected by AND, the type of _m must belong to the intersection of the sets {a_module, c_module}, {c_module, p_module} and {a_module, p_module} which is empty. That means the variable _m can never have any type as specified in the process model.

In order for _m to have a valid type, several possibilities exist. For example, we can change one AND connector in the expression to an OR connector. We can change the destination of the link "has_module" to module. In this case, _m will have the unique type p_module. This second modification implies that the type-checker 2 takes into account the notion of subtype.

4.3 Logic Checker

The role of logic checking is to detect contradictions in logic expressions. The kind of expressions the solver has to deal with are expressions of the first order predicate calculus (not limited to Horn clauses). The checker is just an implementation of the general resolution principle. However, as expressions can contain predicates predefined in the MASP/DL, also called meta-predicates, which have their own semantics (e.g. IS_A_LINK_TYPE(*t : LINK_TYPE*) is a meta-predicate asserting that *t* is a link type of the type *LINK_TYPE*), the algorithm used by the checker is extended with a set of "rewriting rules" which describe the semantics of these predicates. These rewriting rules allow clauses to be rewritten efficiently according to the semantics of the predicates. Rewritten clauses are then treated as others by the general resolution rule. Another point is that as we need the results in a short delay (as permitted by humans), the resolution is limited to a certain number of inference steps, depending on the size of the expression to be checked. Due to this limitation and mainly to the semi-decidable nature of first order logic, the checker cannot always give a firm (affirmative or negative) result when it stops its resolution.

4.4 Importation

Another feature of the tool is that it takes into account the importations of object/operator types. In fact, object and operator types can be shared by several MASPs via importation. For example, an object type can be imported from another MASP and used locally by, say, operator types. However, if the imported object type does not exist, i.e. if it is not defined in the indicated MASP nor imported from another MASP, there is an inconsistency. This kind of inconsistency is detected by the tool. Importations can be nested, i.e. an imported object/operator type in a MASP can be imported again by another MASP. The tool takes into account this point by considering the transitive closure of the importations.

5. Future work

While the tracking of A and B class inconsistencies is efficient and deterministic, the tracking of logic contradictions may be very long. The reason for this is the complexity of the problem and the combinatorial nature of the technique used. We are looking for other more efficient techniques to deal with this problem.

Other work concerns the checking of several related MASPs. As shown in the second section, a process model may be organized into a hierarchy of several MASPs. The presentation above only concerns the inconsistency checking of a single MASP. However, even if each MASP is locally consistent, inconsistencies may still exist among a hierarchy of MASPs. In order to track this kind of inconsistency, more knowledge about the hierarchy is needed.

6. Conclusion

Software process modeling is an important advance in Software Engineering. The verification of the consistency of software process models is becoming more and more critical. Tools for this verification would be very helpful. In this paper, we have presented such a verification tool: ITFM. The tool is capable of tracking syntactic errors, semantic inconsistencies, typing problems and logic contradictions in MASPs enforcing the partial correctness of MASPs. The idea of treating general typing problems as constraint problems, which was developed in our early research, was partially prototyped and validated in the ITFM. Choosing Prolog as prototyping language has facilitated our task. We believe that the techniques explored and prototyped in the ITFM could also be useful and usable in other verification tools for process models. Finally, the ITFM itself is currently being used by the ALF project consortium to verify various process models. It has proven indispensable especially when MASPs are large and complicated. To our knowledge, this is the first tool of its kind designed for the static checking for process models.

Acknowledgements: The authors would like to thank all the members of the ALF consortium: GIE Emeraude (France), CSC (Belgium), Computer Technologies Co. (Greece), Grupo de Mecanica del Vuelo, S.A. (Spain), International Computers Limited (United Kingdom), Cerilor (France), University of Nancy-CRIN (France), University of Dortmund-Informatik X (Germany), Catholic University of Louvain (Belgium) and University of Dijon-CRID (France).

References

[Aho et al. 86] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers-Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1987.

[Benali et al. 89]. K. Benali et al., The Presentation of the ALF Project, In *Proc. of Int. Conf. on System Development Environments and Factories (SDE&F'89)*, Berlin, May 1989. Pitman Publishing, London, March, 1990.

[Boudier et al. 88] G. Boudier, F. Gallo, R. Minot & I. Thomas, An Overview of PCTE and PCTE+. In *Prco. of the 3rd ACM Symposium on Practical Software Develpment Environments,* Boston, Nov. 1988.

[Chabrier et al. 89] J.J. Chabrier, J.K. Hao and F. Trousset, MASP Inconsistency Tracker Requirements. *Technical Report,* ALF/DIJ-JJC/CT-4, Issue 2.0, CRID, Dijon, Oct. 1989.

[Chabrier et al. 91] J. Chabrier, J.J. Chabrier & F. Trousset, Résolution efficaces d'un problème de satisfaction de contraintes: le million de reines. In *Proc. of the 11th Workshop on Expert System & TheirsApplications*, Avignion, France, May 1991.

[Dowson 91] M. Dowson, Software Process Themes and Issues. In *Proc. of the First European Software Process Modeling Workshop*, pp63-72, Milano, Italy, May 1991.

[Giffiths et al. 89] Ph. Griffiths, Ph. Jamart, A. Legait et D. Oldfield, The ALF Approach to Process Modeling. In *Proc. of ESPRIT Conference 1989*. Brussels, 1989.

[Hao & Chabrier 90] J.K. Hao & J.J. Chabrier, A Finite Domain Constraint Solver and its Implementation in Prolog. In *Proc. of the 5th Rocky Mountain Conference on Artificial Intelligence,* pp326-332, Las Cruces, New Mexico, June 1990.

[Hao & Chabrier 91] J.K. Hao & J.J. Chabrier, A Modular Architecture for Constraint Logic Programming. In *Proceedings of the 19th ACM Computer Science Conference (ACM CSC'91)*, San Antonio, Texas, March 1991.

[Keller 88] K.I. Kellner, Representation Formalisms for Software Process Modeling. In *Proc. of the 4th Int. Software Process Workshop,* Moretonhamstead, Devon, UK, May 1988.

[Lehman 87] M.M. Lehman, Process Models, Process programming, Programming Support. In *Proc. of the 9th Int. Conf. on Software Engineering*, pp14-16, Monterey, CA, March 1987.

[Meier et al. 88] M. Meier, G. Macartney, P.A. Tsahaheas, D.H. De villleneuve et D. Chan, *Sepia User Manuel,* TR-LP-38, ECRC, Munich, September 1988.

[Osterweil 87] L. Osterweil, Software Processes are Software Too. In *Proc. of the 9th Int. Conf. on Software Engineering*, pp 1-13, Monterey, CA, March 1987.

[Penedo 86] M.H. Penedo, Prototyping a Project Master Database for Engineering Environment. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Sym. on Practical Software Development Environments*, Pala Alto, California, Dec. 1986, *SIGPLAN Notices* Vol. 22, No. 1, Jan. 1987.

[Trousset 92] F. Trousset, Study of methods of consistency controls based upon resolution techniques: application to software modeling. (in French) Forthcoming *Ph.D Thesis*, University of Dijon, March 1992.