Supporting Impact Analysis and Change Propagation in Software Engineering Environments

Jun Han
Peninsula School of Computing and Information Technology
Monash University, McMahons Road, Frankston, Vic 3199, Australia
phone: +61 3 99044004, fax: +61 3 99044124, e-mail: jhan@monash.edu.au

Abstract

Impact analysis and change propagation are among the major issues of software change management. In this paper, we introduce an approach to providing impact analysis and change propagation support as an integral part of software engineering environments, so that they can be applied during both software development and maintenance. In this approach, the activities are carried out on the original representation of software artifacts in the environment, rather than on a separate system model extracted. This very fact enables automated direct support for the process of actually carrying out and propagating the changes. Dependences and properties of software artifacts are used for impact analysis to achieve increased flexibility and expressiveness. Both impact analysis and change propagation are a combination of guided user intervention and automatic processing based on codified change patterns and propagation rules.

1. Introduction

The development and maintenance of a software system involves a large number of software artifacts, capturing the system's requirements, designs, implementations, testing suites, maintenance records, and so on. Traditionally changes to a software system are referred to those related to software maintenance. During the initial development of the system, however, there are also on-going requirements for change due to changed requirements, error correction and development improvement. Therefore, the requirement for change is an inherent characteristic of the software development and maintenance process. In any case, the required changes are eventually reflected in changes to the software artifacts involved. Since the software artifacts are logically related to each other, an initially proposed change may involve a large number of modifications to various artifacts. As such, the process of software changes needs to be managed and, if possible, assisted by automated tools.

The issues of software change management range from identifying the need for change, assessing the impact of a proposed change in the entire system, carrying out the initial and consequent modifications, managing the versions of the changed artifacts, to collecting change-related data [15]. Over the years, we have seen efforts addressing these issues. They include process models for change management such as Prism [15], frameworks for comparing impact analysis methods such as [2], specific techniques for change impact analysis and change propagation such as program slicing [18, 12] and program dependence graphs [16, 14, 11], and approaches to tool support for change management [13, 4, 7, 5].

In this paper, we address the tool support issues for software change management, in particular for impact analysis and change propagation, in the context of a (generic) software engineering environment [19, 8, 9]. In our approach, the kernel of the software engineering environment, augmented with specific features for change management, can be seen as a generic tool for change management. The policies, models and methods for change management in a particular application context (e.g., a specific software development and maintenance project) can be codified to specialise the environment kernel, so that the resulting environment provides the specific change management support (as well as other support) as required by the application context. As such, this approach recognises the varied needs of different application contexts, which may be due to the nature of the particular system to be developed, the different development methods and languages to be used, and so on. In particular, our approach has the following characteristics:
1. The original representation of the software system (artifacts and dependences) as developed and maintained in the environment is directly used for change management support, so that there is no information extraction or re-formulation required.

2. In addition to artifact representations and dependences, artifact properties and change patterns are also used for change impact analysis, which has greatly increased the flexibility and expressiveness of the approach.

3. The use of the original system representation allows automated direct assistance to change propagation (i.e., actually carrying out the modifications) based on the impact analysis results and additional change propagation rules.

4. Both change impact analysis and change propagation are a combination of automatic processing (based on change patterns or propagation rules) and guided user intervention.

These are in clear contrast to most existing approaches, where the impact analysis activities are usually performed based on an extracted system representation involving only artifact structures and dependencies (without their contents), and as such the actual change propagation process is not directly supported.

In general, our approach provides support for system representation, impact analysis, and change propagation. They are discussed in sections 2, 3 and 4 respectively. In section 5, we present an example to demonstrate the use of our approach. After discussing related work in section 6, we conclude in section 7.

2. System representation

Fundamental to any software change management system is the representation of software artifacts and their dependences. In the context of a proposed change, the software artifacts are objects of modification, and their dependences are among the means to realise the impact and propagation of modifications. We also use the consistency properties of software artifacts and dependences for change management. While not usually addressed in the literature as such, the use of consistency properties provides additional flexibility and expressiveness in specifying and managing change activities (see below).

In our approach, the change management activities are performed on the actual software artifacts and their dependences developed and maintained in the software engineering environment. The requirements for system representation are largely met by the general representation facilities of the environment. Here we only give a brief overview of these facilities, and further details can be found in [8, 11, 9]. Examples can be found in section 5.

Artifact representation. Artifact representation in our environment has the following characteristics which are particularly relevant to change management:

1. The representation is based on syntax trees and uses an augmented EBNF notation for specification [8]. Therefore, it is fine-grained, and facilitates flexible references to various levels of artifact components. If desired, component types at intermediate levels can be identified as atomic for change management purpose. As such, it satisfies the flexibility requirements when referring to artifact components during change management [2].

2. The artifacts and their components are typed in the representation. As such, it will better support the change management capabilities [6].

Dependence definition. Dependences related to change management can be captured using the relationship definition mechanism provided by the environment kernel, where a relationship connects one artifact (component) to another artifact (component) and possibly with its own content and attributes [8]. In fact, the dependences should be among the relationships that are required, introduced and maintained during artifact construction and manipulation. As such, there is no separate dependence representation required for change management. On the other hand, we may exclude certain relationships from consideration with respect to change management.

As in the case of artifact representation, this representation of dependences has the following major characteristics relevant to change management:

1. The dependences can be defined accurately, due to the fine-grained artifact representation. This will enable us to more precisely locate dependence-based consequent modifications.

2. The dependences are typed, which provides the necessary expressiveness required by impact analysis and change propagation.

Consistency properties. Most of the literature on software change management addresses change impact analysis and propagation in terms of dependences, in view of modifying an existing set of software artifacts.
This is not surprising as change management is primarily about changing existing software artifacts. In order to implement the changes, however, it is often necessary to introduce new artifacts such as a new module, or remove existing artifacts which are no longer required. In general, the consequences of introducing or removing an artifact or dependence should also be taken into account during impact analysis and propagation. Dependences alone do not adequately facilitate identification of the impact related to the introduction and removal of artifacts and dependences. Consistency properties provide an additional mechanism to deal with the issue.

As an example, consider the following property of a given development methodology:

"a design module in a design document must be related to one or more implementation modules in the corresponding implementation document."

Accordingly, the introduction of a design module into the design document may possibly imply the introduction of one or more new implementation modules and associated dependences between the implementation modules and the design module.

In general, properties related to change management can be stated using the environment kernel’s property specification mechanism [8, 11]. Again, properties that are not relevant to change management can be excluded from consideration.

3. Impact analysis

An essential part of software change management is to predict the system-wide impact of a change request before actually carrying out modifications to the system – impact analysis, so that appropriate decisions related to the change request can be made, such as planning, scheduling and resourcing. Impact analysis starts with a set of initial modifications to some software artifacts and possibly their dependences, directly reflecting the change request. For a given type of modification, potential consequent modifications can be identified along relevant dependences and according to relevant consistency properties. Whether a potential modification is a required modification can be decided by the system (based on a codified change pattern) or by the user (interactively). Then the required consequent modification is regarded as a new initiating modification to identify further consequent modifications. This process continues until no further consequent modifications can be identified. In this section, we discuss in detail the relevant issues.

Objects and types of modification. The object of a modification is a software artifact, an artifact component, or a dependence. The types of possible modifications for an object are update, introduction and deletion. While introduction and deletion are self-explanatory, update is applied to an existing object and does not change its type. In general, therefore, a modification is characterised by a pair, \((obj, m\_type)\), where \(obj\) is the object of modification and \(m\_type\) is the type of modification. Sometimes, we also write this characterisation as \((obj : obj\_type, m\_type)\) or \((obj\_type, m\_type)\) to emphasise the object type.

Consequent modifications. For an initiating modification \((obj, m\_type)\), the potential objects of immediate impact are the objects that are involved with \(obj\) in dependences, properties, derived dependences, or derived properties. The cases based on derived dependences or properties capture the situations where a potential consequent impact can not be directly captured by a defined dependence or property. In general, these situations can be captured by a change pattern (see below for detailed definition).

As mentioned above, the determination of a potential consequent modification can be carried out through an interactive process between the user and the system. That is, the system prompts the user, and the user decides (1) whether or not the potential modification is a required modification, and (2) if so, the type of the modification. When a potential modification is identified as a required modification, it is linked to the initiating modification for traceability purpose. For certain cases, however, this process can be automated by codifying change patterns into the support environment.

Automatic impact analysis. A change pattern takes the following format:

\[
\{mod_{11}, mod_{12}, \ldots\} \rightarrow \{mod_{C1}, mod_{C2}, \ldots\} : expB;
\]

where

1. \(mod_{ij}(j = 1, 2, \ldots)\) describes a pattern of initiating modifications, and takes the form of \((obj_{ij} : obj\_type_{ij}, mod\_type_{ij})\) with \(obj_{ij}\) being an object variable, \(obj\_type_{ij}\) an object type (i.e., an artifact (component) type or dependence type and \(mod\_type_{ij}\) a modification type;

2. \(mod_{Cj}(j = 1, 2, \ldots)\) describes a pattern of consequent modifications, and takes the form of \((obj_{Cj} : obj\_type_{Cj}, mod\_type_{Cj})\) with \(obj_{Cj}\) being an object variable, \(obj\_type_{Cj}\) an object type and \(mod\_type_{Cj}\) a modification type;
3. \( \text{exp}_B \) is a Boolean expression about the object variables concerned:

\[ \text{exp}_B(o_{j1}, o_{j2}, \ldots, o_{jC}, o_{j2'}, \ldots) \]

The pattern states that modifications of type \( \text{mod} \_\text{type}_{2j} (j = 1,2, \ldots) \) on object \( o_{j1} (j = 1,2, \ldots) \) of type \( \text{obj} \_\text{type}_{2j} (j = 1,2, \ldots) \) will result in modifications of type \( \text{mod} \_\text{type}_{2c} (j = 1,2, \ldots) \) on \( o_{j2'} (j = 1,2, \ldots) \) of type \( \text{obj} \_\text{type}_{2c} (j = 1,2, \ldots) \) if and only if the Boolean expression \( \text{exp}_B \) holds. The Boolean expression usually states the relationship of the objects involved. During impact analysis, a potential modification will be resolved automatically when it matches a change pattern.

A potential change pattern for impact analysis based on derived dependences and properties is described in the following form:

\[ \{ \text{mod}_{11}, \text{mod}_{12}, \ldots \} \rightarrow \{ \text{mod}_{C1}, \text{mod}_{C2}, \ldots \} : \text{exp}_B \]

Note that normal change patterns resolve potential consequent modifications automatically while potential change patterns only identify potential consequent modifications to be resolved by the user.

**Summary.** For a potential consequent modification (identified based on either dependences, properties, or derived dependences and properties), in general, the environment will first try to find a matching change pattern. If found, the pattern is applied and the potential consequent modification is resolved. Otherwise, the scenario is presented to the user for resolution.

Upon completion of impact analysis, we are left with a set of required modifications together with the impact relationships between them. In fact, they form a directed acyclic graph (DAG) structure with the nodes denoting the modifications and the edges denoting the impact relationships, which we call the impact graph.

4. **Change propagation**

The next step following impact analysis is to actually carry out the identified modifications - change propagation. Ideally, the modifications should be made in an order dictated by the impact relationships. On the other hand, it is also possible to carry out the modifications independently and even concurrently without following the impact relationships, as long as the requirements for the modifications concerned are clear.

The consistency among the software artifacts can be re-established after the modifications. In the following discussion, we concentrate on a process of change propagation according to the impact relationships.

**Interactive change propagation.** At a given time, in general, the modifications concerned can be divided into four categories:

\( (\text{past}, \text{current}, \text{next}, \text{future}) \)

where the past modifications are the ones that have been completed, the current modifications are those that are being carried out, the next modifications are the ones that can be selected next according to the impact relationships, and the future modifications are the rest. At the beginning, the sets of past and current modifications are empty, the set of next modifications contains the initial modifications, and the set of future modifications contains all the identified consequent modifications. At a given time, the set of next modifications may contain more than one modification, and therefore more than one modification can be selected as current modifications if so desired.

The change propagation process is as follows:

1. Upon on selection by the user, an identified modification from the set next will become a current modification. The sets current, next and future are adjusted accordingly.

2. The user will then carry out the selected modification as required.

3. When a current modification is completed, it will be moved to the set past.

4. The selection-modification-completion cycle continues until all the identified modifications are completed (i.e., all in the set past).

After a modification being moved into the set past, the user may later want to make further updates to the object concerned, and the modification sets should be adjusted accordingly.

In general the environment will maintain the impact graph and the related categories (sets) of modifications. It is also responsible for realising the adjustments to these categories as the change propagation process progresses. On the other hand, the user is responsible for making the actual changes. As such, the above change propagation process is an interactive process.

**Automatic change propagation.** For well-defined cases, the impact of certain modifications can be propagated automatically according to codified propagation rules. In general, there are two kinds of change propagation rules. The first case is that certain modifications are fully realised automatically as the consequence of...
other modifications. This type of rules take the following form:

$$\{mod_{i1}, mod_{i2}, \ldots\} \rightarrow \{mod_{C1}, mod_{C2}, \ldots\} : \text{exp}_{B} \text{ by action;}$$

which means that the consequent modifications $mod_{Cj}(j = 1, 2, \ldots)$ are realised by the action as the consequence of completing all initiating modifications $mod_{ij}(j = 1, 2, \ldots)$. The action is a code segment or function call that can access the objects involved. The firing of this rule is subject to the satisfaction of the Boolean expression $\text{exp}_{B}$.

Note that the above propagation rule is similar to a change pattern, except that it has an additional action. In general, if the scenario captured by a propagation rule is the same as that captured by a change pattern, the action can be specified together with the change pattern.

The second type of propagation rules are to deal with cases where only part of the consequent modifications can be done automatically due to the effect of some initiating modifications. This type of rules are particularly useful when the granularity of impact analysis is relatively coarse (but change propagation can be carried out at a lower granularity). The propagation rules of this type take the following form:

$$(obj_{i1} : obj_{type_{1}}, \ldots) \rightarrow (obj_{C1} : obj_{type_{C1}}, \ldots) : \text{exp}_{B} \text{ by action;}$$

where $obj_{ij}(j = 1, \ldots)$ are objects involved in the initiating modifications, $obj_{ij}(j = 1, \ldots)$ are objects involved in the consequent modifications, and $\text{exp}_{B}$ is the condition to be evaluated before activating the action to realise the partial effects of the consequent modifications. The triggering of such propagation rules will not cause any category change for consequent modifications, because not all impact effects are realised. For reference purpose, however, the consequent modifications are annotated with the rules to indicate the partial realisation of the impact.

Summary. In general, all the relevant propagation rules are fired (if possible) whenever a modification completes (no matter that it is due to the interactive or automatic process) and the rules match. As such, automatic and interactive change propagation works in an interwoven fashion.

5. An example

In this section, we present an example to demonstrate how our software change management approach can be applied. For simplicity, we consider a partial software development methodology that involves a design document developed in the Booth method [6] and its corresponding implementation document in C++, with certain customisation to ascertain a clear development process. Furthermore, we will show only the relevant formalisation of the methodology in a particular application context, i.e., a simplified version of the weather monitoring system presented in [6].

The weather monitoring system is to provide automatic monitoring of various weather conditions, and in particular measuring wind speed and direction, temperature, barometric pressure and humidity (further details can be found in [6]). As Booth suggested, after the system has been developed there may be a change requirement to measure rainfall as well, which we use as the change request in this case study.

5.1. System representation

A design document developed in the Booth methodology may contain class diagrams, state transition diagrams, and a number of other types of diagrams. Here, we only consider a class diagram capturing the architecture of the software system, and state transition diagrams capturing the dynamic behavior of individual classes of the class diagram. Figure 1 shows a partial design of the weather monitoring system, containing the class diagram and an illustrative state transition diagram [6]. The class diagram involves ten ordinary classes and three abstract classes, with “association”, “has” and “inheritance” relationships between them (please ignore the class RainfallSensor for the moment). The state transition diagram IMstd details the class InputManager’s state changes upon pressing different types of user input keys (not shown). The relationship between InputManager and IMstd is captured by a hasSTD relation. Some high-level type rules (in an EBNF form) for the representation of this design are

$$\text{DesignDoc} = \text{ClassDiagSpec} \{ \text{STD} \};$$
$$\text{ClassDiagSpec} = \{ \text{Class} \} \{ \text{AbsClass} \};$$

The relevant relations/dependences can be defined as

$$\text{association} = \langle \text{Class} | \text{AbsClass}, \text{Class} | \text{AbsClass} \rangle;$$
$$\text{has} = \langle \text{Class} | \text{AbsClass}, \text{Class} \rangle;$$
$$\text{inheritance} = \langle \text{Class} | \text{AbsClass}, \text{Class} | \text{AbsClass} \rangle;$$
$$\text{hasSTD} = \langle \text{Class}, \text{STD} \rangle;$$

Note that a relation type rule defines a relation type (on the right hand side of =) using types of the document components to be connected (in arrow brackets.
The implementation document is a C++ program, in which there is a C++ class for each design class. Among these C++ classes, there are relationships/dependences mirroring the association, has and inheritance relationships between their corresponding design classes. In addition, other components may present, containing entities such as the main function and simple type declarations. A particular type declaration is the one enumerating the different types of sensors [6]:

```c
//Enumeration of sensor names
enum SensorName (Direction, Speed, Temperature, Humidity, Pressure);
```

which can be used by various implementation classes. In particular, ImpDisplayManager (i.e., the implementation class corresponding to DisplayManager) uses it to discriminate the display functionality according to the sensor type of the data to be processed. So there is a typeToUse dependence between the above type declaration and the implementation class ImpDisplayManager. Besides, we may introduce a relationship of type enumToImpClassH between the above enumerated type declaration and the root of the inheritance hierarchy in the implementation document (i.e. ImpSensor) to indicate that the values in the enumerated type declaration reflect the concrete classes on the inheritance hierarchy.

Some type rules for the implementation document are

```c
ImpDoc = CppProgram;
CppProgram = { ImpClass } { ImpAbsClass }
MainFunc { TypeDec } ...
TypeDec = EnumDec | ...
```

There are also the following types of
relations/dependences:

\[ \text{impAssociation} = \]
\[ <\text{ImpClass} \mid \text{ImpAbsClass} \mid \text{MainFunc}, \]
\[ \text{ImpClass} \mid \text{ImpAbsClass} >; \]

\[ \text{impInheritance} = \ldots \]

\[ \text{impHas} = \ldots \]

\[ \text{enumToImpClassH} = \]
\[ <\text{EnumDec}, \text{ImpClass} \mid \text{ImpAbsClass}>; \]

\[ \text{typeToUse} = <\text{TypeDec}, \]
\[ \text{ImpClass} \mid \text{ImpAbsClass} >; \]

Between the entities of the design document and the entities of the implementation document, there exist different types of relationships. In particular, there is a \text{designToImp} relationship between the entire design document and the entire implementation document, and a \text{hasImp} relationship between a design class and its corresponding implementation class. The relevant relation type rules are

\[ \text{designToImp} = <\text{DesignDoc}, \text{ImpDoc}>; \]

\[ \text{hasImp} = <\text{Class}, \text{ImpClass} > | \]
\[ <\text{AbsClass}, \text{ImpAbsClass}>; \]

As an example consistency property, consider the requirement that there must be an implementation class for each design class in the development. This implies that a mathematical relation formed from all the \text{hasImp} relationships is total relative to the set of all design classes:

\[ \text{propHasImp} = \text{total}(\text{hasImp}, \text{Class}); \]

where \text{total} is a primitive predicate.

5.2. Impact analysis

To work out the impact of the suggested change request about the weather monitoring system, we first need to identify the initial modifications required:

1. \text{introRS}: A new design class \text{RainfallSensor} needs to be introduced (see Figure 1). This modification is characterised by or has the type of, \text{(Class, introduction)}.

2. \text{introRSInh}: The new design class inherits from the abstract class \text{HistoricalSensor}. This modification has the type of \text{(inheritance, introduction)}.

3. \text{introRSHas}: The design class \text{Sensors} will have instances of the new design class. This modification has the type of \text{(has, introduction)}.

4. \text{changeIMstd}: Since there is a new key in the user interface for Rainfall, the state transition diagram \text{IMstd} will need to be changed. This modification has the type of \text{(STD, update)}.

Note that the modifications are provided with a name for reference convenience.

The initial modifications are all relevant to the design document. As the first step, let us examine if there are any consequent impacts in the design document. First of all, the introduction of the new design class (i.e. \text{introRS}) does not have direct impact in the design document. Since the new class is not directly connected to any artifact component, there is no change pattern required to indicate the above effect.

Next, the introduction of the inheritance relation (i.e. \text{introRSInh}) does not have direct impact on the parent class in the design document. To identify this null-effect automatically, we have the following change pattern:

\[ \{(i : \text{inheritance}, \text{introduction}) \rightarrow \{(\theta, -) : <\ast, i, b >; \}
\]

where "." indicates that no modification is required, and \(<\ast, i, b >\) indicates that the relationship \(i\) relates an artifact component to \(b\).

As to the introduction of the \text{has} relation (i.e. \text{introRSHas}), a potentially impacted object in the design document is the \text{Sensors} class. When the case is presented to the user, the user decides there is no change required, i.e., an interactive impact analysis step.

As to the update to \text{IMstd} (i.e. \text{changeIMstd}), the candidate object for potential direct impact is the \text{hasSTD} relationship. In fact, no modification is required for \text{hasSTD}, as captured by

\[ \{(a : \text{STD}, \text{update}) \rightarrow \{(r : \text{hasSTD}, -) : <\ast, r, a >; \}
\]

In general, therefore, there is no consequent modifications in the design document. However, there are consequent modifications in the implementation document following the relevant dependences and properties. First of all, the introduction of the new design class \text{RainfallSensor} will cause the introduction of a corresponding implementation class \text{ImpRainfallSensor} and the introduction of a \text{hasImp} relationship between the two classes. This impact can be captured by the following change pattern:

\[ \{(a : \text{Class}, \text{introduction}) \rightarrow \{(b : \text{ImpClass}, \text{introduction}), \]
\[ (r : \text{hasImp}, \text{introduction}) : <\ast, r, b >; \]

178

Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97)
0-8186-7840/97 $10.00 © 1997 IEEE
This change pattern allows the support environment to carry out the above impact analysis automatically. We give the two consequent modifications the following names respectively: introImpRS and introRSHasImp.

Instead of the above change pattern, we may rely on the consistency property propHasImp to analyse the impact. The introduction of the design class affects this property, and consequently the system will present to the user all the possibly affected objects according to the property, i.e., a new implementation class and a new hasImp relationship between the classes. The user then confirms the need for their introduction.

The introduction of the inheritance and has relationships in the design implies the introduction of the corresponding relationships in the implementation. This impact analysis can be done by the system according to the following change patterns:

\[
\begin{align*}
&\{(\text{r}_1 : \text{inheritance}, \text{introduction})\} \rightarrow \\
&\{(\text{r}_2 : \text{impInheritance}, \text{introduction})\} : \\
&\langle a_1, r_1, b_1 \rangle \land (a_1, a_2) \in \text{hasImp} \land \\
&(b_1, b_2) \in \text{hasImp} \land \langle a_2, r_2, b_2 \rangle;
\end{align*}
\]

\[
\begin{align*}
&\{(\text{r}_1 : \text{has}, \text{introduction})\} \rightarrow \\
&\{(\text{r}_2 : \text{impHas}, \text{introduction})\} : \\
&\langle a_1, r_1, b_1 \rangle \land (a_1, a_2) \subset \text{hasImp} \land \\
&(b_1, b_2) \in \text{hasImp} \land < a_2, r_2, b_2 >;
\end{align*}
\]

We name these two consequent modifications as introImpRSInh and introImpRSHas respectively.

The addition of the new implementation class to the inheritance hierarchy causes change to the enumerated type SensorName. To identify this impact, we provide the following potential change pattern:

\[
\begin{align*}
&\{(\text{a} : \text{ImpClass}, \text{introduction})\} \rightarrow \\
&\{(\text{b} : \text{EnumDec}, \text{update})\} : \\
&(b, c) \in \text{enumToImpClass} \land \\
&(c = a) \lor (a, c) \in \text{impInheritance}^+);
\end{align*}
\]

As such, the introduction of the new implementation class ImpRainfallSensor will lead to the identification of the SensorName declaration as a potential candidate for update. When presented to the user, this impact is confirmed. We name this modification changeEnum. Note that we formulate the above change pattern as potential (\(\sim\)) that is, there is no automatic resolution because not every introduction of an implementation class will lead to the update of an enumerated type declaration.

The consequence of changeEnum would be the impact on places using SensorName. Following the typeToUse dependences, the implementation class ImpDisplayManager is confirmed by the user as an actual impact point for update to reflect the specific display requirement of ImpRainfallSensor. We call this modification changeImpDM.

According to our development methodology, adding a has relationship to a design class will lead to changes to the corresponding implementation class. This can be captured by the following change pattern:

\[
\begin{align*}
&\{(\text{h} : \text{has}, \text{introduction})\} \rightarrow \\
&\{(\text{c} : \text{ImpClass} \cup \text{ImpAbsClass}, \text{update})\} : \\
&(a, c) \in \text{hasImp} \land < a, h, * >;
\end{align*}
\]

According to this pattern, ImpInputManager is automatically identified as requiring update due to the change to IMstd. We name this new modification as changeImpIM.

In summary, we have four initial and eight consequent modifications. Together with their impact relationships, they form a DAG structure (see Figure 2).

5.3. Change propagation

When actually carrying out the above changes, the support system will provide assistance to the user as to what the next logical modifications are and the major reasons for a modification (as indicated by the impact relationships). In addition, we may introduce an automatic propagation rule to let the system automatically introduce a class skeleton for ImpRainfallSensor and the relevant hasImp relationship, and add to the set of modifications a new update modification on ImpRainfallSensor for filling in its details. This can be achieved by adding an action to a relevant change pattern defined earlier:

\[
\begin{align*}
&\{(\text{a} : \text{Class}, \text{introduction})\} \rightarrow \\
&\{(\text{mod} : (\text{b} : \text{ImpClass}, \text{introduction}), \\
&(\text{r} : \text{hasImp}, \text{introduction})) \land < a, r, b > \} \text{ by} \\
&\text{begin} \text{b} = \text{introImpCluss}(); \\
&\text{introRel}(a, b, \text{hasImp}); \\
&\text{introMod}(b, \text{"update"}, \text{mod}); \\
&\text{end;}
\end{align*}
\]
where \textit{introImpClass} is a function for introducing a skeleton implementation class, \textit{introRel} is a primitive function for introducing a relationship, and \textit{introMod} is a function for introducing a new modification, \((b, \text{update})\), which is related to the initiating modification \(mod\). We name the new modification \textit{changeImpRS} (see Figure 2).

Another example would be a propagation rule for automatically introducing the inheritance relationship in the implementation document following the introduction of the inheritance relationship in the design document and the introduction of the implementation class \textit{ImpRainfallSensor}:

\[
\{(r_1: \text{inheritance}, \text{introduction}),
\ (a: \text{ImpClass}, \text{introduction})\} \rightarrow
\{(r_2: \text{impInheritance}, \text{introduction})\}:
\begin{align*}
&\ angle a_1, r_1, b_1 > \land (a_1, a) \in \text{hasImp} \land \\
&\ (b_1, b_2) \in \text{hasImp} \land < a, r_2, b_2 >
\end{align*}
\text{by \textit{introRel}(a, b_2, "impInheritance");}
\]

Note that this propagation rule is similar to an earlier change pattern, but they are different. In particular, there is an additional precondition for the rule to be fired, i.e., the completion of an additional initiating modification regarding the introduction of the implementation class. This indicates that a propagation rule may contain more initiating modifications than its corresponding change pattern. Also note that this rule is not a change pattern and is not used in impact analysis. Similarly we can define a propagation rule to introduce the has relationship in the implementation document automatically.

6. Related work

Traditional impact analysis approaches based on program slicing (e.g., [18, 12]) and program dependence graphs (e.g., [16, 14]) concentrate on impact analysis of program code. They mostly exploit the data dependence and control dependence as implied in the language definitions [3]. As such, other dependences such as the one between the enumerated type declaration and the concrete classes on the sensor inheritance hierarchy in the given example can not be captured. Besides, most of these approaches perform impact analysis on an extracted system representation, and therefore usually do not provide direct support for the change propagation process.

The approaches suggested in [1, 7] concentrate on assistance to change propagation, instead of impact analysis. They achieve this through the use of consistency rules. In [1], the violation of a consistency rule by a modification to the program is reported to the user, and the user is supposed to take action. In [7], message passing is used to propagate changes in order to enforce consistency rules, and to report consistency violations. In [4], an approach based on expert systems is proposed. Software dependences in terms of change are codified as rules in the expert system for impact analysis. Rules are also used to provide advice for change propagation. However, it is not clear how the system would directly support the actual change propagation process.

In [5], an impact analysis tool is reported. It is based on an extracted dependence graph model of system representation, where the nodes and edges are typed and are used to model artifacts and their dependences. It supports the specification of actual and potential im-
impact analysis rules in terms of the system representation. Impact analysis according to the rules is an automatic process with potential impacts marked through a "weight" system. The potential impacts are then interactively validated by the user. The effectiveness of the automatic impact analysis solely based on codified rules in this approach is unclear. This is in clear contrast to our approach, where the dependencies, properties and change patterns are all used for impact analysis to give a broader coverage and on the other hand the change patterns are also used to reduce the number of potential impacts for user validation.

7. Conclusions

In this paper, we have presented an approach to software change impact analysis and propagation in the context of a software engineering environment. In clear contrast to existing approaches, our approach uses the environment representation of software artifacts and their dependences. As such, there is no extracted, separate system representation required. Impact analysis and change propagation are performed directly on the actual (typed and fine-grained) software artifacts and dependences, and can be used during both software maintenance and initial system development. Besides, other environment facilities are also available for use during impact analysis and change propagation, including those for defining high-level operations and presentation views [19, 9].

In addition to dependences, we have used properties about software artifacts and dependences for impact analysis. This has greatly increased the flexibility and expressiveness of the approach, and made it possible to provide the capabilities of both traceability and dependence analysis based approaches [3]. Consequently, our approach is able to handle both source code and other types of software artifacts. In particular, our approach is able to identify impacts concerning the introduction and deletion of software artifacts and dependences, which is not usually addressed by existing approaches. The impact analysis process in our approach is a combination of automatic application of codified change patterns and interactive confirmation of potential impacts.

The use of the system representation in the environment has enabled automated direct assistance to change propagation. This is in contrast to most existing approaches, where impact analysis is performed on an extracted system model and no direct propagation support is provided. Again, the change propagation process is a combination of automatic propagation based on codified rules and interactive user guidance based on the impact analysis results.

A prototype experiment of the proposed approach is currently being undertaken in a C++ programming environment developed using the Synthesizer Generator [17]. Its full-scale implementation in our software engineering environment is to follow. Some of the issues for further investigation include incorporation of change rationales, integration with configuration and version management capabilities [10], and management of changes to artifact properties, change patterns and propagation rules.

Acknowledgment. The author would like to thank Hardeep Singh and the anonymous reviewers for their valuable comments on earlier drafts of this paper. The work reported here was partially funded by the Australian Research Council.

References


182