

Semester Project Report

Integrating Simplify into Jive

Yoshimi Takano
ytakano@student.ethz.ch

Software Component Technology Group
Department of Computer Science
ETH Zurich

<http://sct.inf.ethz.ch/>

October 2006

Advisors:

Ádám Darvas
Prof. Dr. Peter Müller

The Java Interactive Verification Environment, *Jive*, is a verification system for object-oriented programs on the basis of a partial-correctness Hoare-style programming logic. The verification of general lemmas that arise during a program proof is delegated and performed by an associated general theorem prover. Currently, the interactive theorem prover *Isabelle* is supported for this purpose. This report describes the integration of the fully automatic theorem prover *Simplify* as an additional associated theorem prover.

Contents

1. Introduction	6
1.1. Jive	6
1.2. Project Task	6
2. Simplify	8
2.1. Terms and Predicates	8
2.2. Axioms and Simplify Sessions	10
2.3. Quantifiers and Triggering Patterns	10
3. Data and State Model Formalization	13
3.1. Typed vs. Untyped Logics	13
3.1.1. Straight Forward Approach	14
3.1.2. The ‘as’ Trick	16
3.1.3. No Type Information at All	17
3.1.4. Approach Taken	19
3.2. Translation	21
4. Simplify Unparser	25
4.1. Terms and Formulas	25
4.2. Implementation	27
5. Workflow and GUI Integration	30
5.1. Interaction with Simplify	30
5.1.1. Implementation	31
5.2. User Preferences	33
5.3. Triggering a Simplify Proof	35
5.4. Future Work	37
6. Conclusion	38

A. Appendix	40
A.1. Terms and Formulas in Simplify	40
A.1.1. Terms	40
A.1.2. Formulas	41
A.2. Terms and Formulas in Jive	42
A.2.1. Terms	42
A.2.2. Formulas	44

1. Introduction

1.1. Jive

The Java Interactive Verification Environment, *Jive* [1, 2], is a verification system that is being developed jointly at University of Kaiserslautern and at ETH Zurich. It is an interactive special-purpose theorem prover for the verification of object-oriented programs on the basis of a partial-correctness Hoare-style programming logic. Currently, Jive operates on *Java-KEx* [9], a desugared subset of sequential Java which contains all important features of an object-oriented language (subtyping, exceptions, static and dynamic method invocation, etc.). Programs can be annotated using *JML* [5] as specification language, which provides concepts such as invariants and pre- and postconditions. Jive is written in Java.

Program proofs in Jive are represented by trees that are built from instances of the axioms and rules of the underlying programming logic. During a program proof, general properties to be proven (called *verification conditions* or simply *lemmas*) arise from applications of rules that manipulate the pre- or postcondition of the examined subgoal without affecting the current program part selection. The verification of such lemmas is delegated and performed by an associated general theorem prover. Currently, the interactive theorem prover *Isabelle* [3] is supported for this purpose.

In order to prove these logical predicates, Isabelle needs a formal data and state model of the target language of Jive. The data and state model describes the language's types, values, objects and object states and acts as interface between Jive and Isabelle. It consists of a program independent part that formalizes, e.g., types and values, and a program dependent part that contains information about the types, fields, etc. declared in a program.

1.2. Project Task

In the current implementation of Jive, the generated verification conditions have to be exported and separately proven in Isabelle. There is no online communication with

Isabelle that would make this interaction more user-friendly. As another drawback, Isabelle is difficult to work with for non-experts and users lacking background information about the underlying data and state model. For instance, a user has to know the name of a helper lemma in order to use it in an Isabelle proof.

The goal of this semester project is to integrate the fully automatic theorem prover *Simplify* into Jive, ultimately aiming at bringing Jive to a higher level of automation and usability. The idea is to have part of the verification conditions proven automatically by *Simplify* before or even instead of exporting them to Isabelle.

The project can be broken down into three main subtasks:

- (i) Translation of the Isabelle formalization of the data and state model for the target language of Jive to the *Simplify* input language.
- (ii) Implementation of a module that unparses formulas from Jive's internal representation to the *Simplify* input language. This unparser is used (1) when generating the program dependent part of the data and state model and (2) to unparse the verification conditions to the *Simplify* input language before they are sent to *Simplify*.
- (iii) Integration of the *Simplify* component into the Jive verification workflow (including GUI modifications).

After a general introduction to *Simplify* in Chapter 2, one chapter is dedicated to each of the above mentioned subtasks where encountered problems and corresponding solutions are discussed. Finally, we present our conclusions in Chapter 6.

2. Simplify

This preliminary chapter aims at familiarizing the reader with Simplify and some of its core concepts. In addition, some terminology is introduced that will be useful in the rest of this paper.

Simplify is a fully automatic theorem prover that attempts to prove first-order formulas. Although originally designed for the Extended Static Checking (ESC) project [6], the prover is interesting in its own right and has been adopted in several other projects related to program verification [11, 12]. For the project at hand, Simplify has been used in its current version 1.5.4 and all comments throughout this report refer to that particular version.

Simplify, written in the Modula-3 language, accepts a sequence of first-order formulas as input and tries to prove each one. Simplify does not implement a decision procedure for its inputs: it may fail to prove a valid formula. However, it is conservative in that it never claims that an invalid formula is valid. Furthermore, Simplify's computations can be bounded, e.g., by a time limit.

For more details about the implementation of Simplify, the reader is referred to the technical report [4].

2.1. Terms and Predicates

Simplify's logic is completely untyped. However, a strong distinction is drawn between *terms* and *predicates*. Terms are expressions that represent values in an underlying value space (*individual values*). Predicates, or *formulas*, are expressions that represent truth values (*propositional values*).

In Simplify, a term is a term constant, a variable, or an application of a function to terms. Simplify provides some built-in term constants, such as “0” and “42”, and some built-in functions, namely “+”, “-” and “*” with the obvious interpretation over the integers. A *ground term* is a term with no variables.

A predicate in Simplify is a predicate constant (**True** or **False**), an application of a built-in predicate symbol to terms, an application of a boolean connective to predicates,

or a quantified predicate. Simplify’s built-in predicate symbols include “=”, “≠” and “<”. Its built-in boolean connectives include “¬”, “∧”, “∨”, “⇒” and “⇔”.

For a more detailed description of terms and predicates please refer to Appendix A.1. As we will see in Chapter 4, there is a mismatch between the term/predicate model in Simplify and the representation of first-order formulas internal to Jive. In what follows, whenever we speak of a term or predicate without further annotation, we mean a term or predicate as defined in the Simplify world.

User-Defined Terms and Predicates

Simplify provides mechanisms by which users can implicitly declare (term) constants, variables and functions. More precisely, symbol names are considered as symbolic constants when they are not bound by a quantifier, and as variables otherwise. Function symbols other than the built-in ones are uninterpreted and implicitly declared at their first use.

On the other hand, Simplify does not allow a user to directly declare new predicate symbols. However, Simplify provides a special built-in term constant **@true** to reflect the predicate constant **True** in the term space. This can be used to model a predicate as a function whose result is equal to **@true** iff the predicate holds for its arguments. So, if we want to introduce a new unary predicate F , say, as

$$\forall x: (F(x) \Leftrightarrow \text{body}),$$

we would use a function f instead and actually write

$$\forall x: (f(x) = \text{@true} \Leftrightarrow \text{body}),$$

essentially replacing every occurrence of $F(x)$ with $f(x) = \text{@true}$. We refer to such a function f as a *quasi-predicate*. For the sake of simplicity, however, we sometimes still use the notation $F(x)$ and call it a user-defined predicate in this report, even if there is actually a quasi-predicate involved¹.

Note that **@true** is a built-in *term* constant, not to be confused with the built-in predicate constant **True**.

¹Actually, Simplify has a feature allowing a user to introduce predicate symbols that are automatically rewritten to quasi-predicates, though involving some heuristic differences. However, this is not relevant for our purposes.

2.2. Axioms and Simplify Sessions

During a Simplify session, the prover maintains a set of *axioms* (also called *background predicates*) that are assumed to be true. The axioms are organized as a stack such that they can be added and removed in a LIFO pattern. Throughout this document, we display axioms with a prefix \textcircled{A} like this:

\textcircled{A} axiom.

For an example Simplify session, consider the following axioms from group theory:

- \textcircled{A} $\forall x: \text{times}(e, x) = x,$
- \textcircled{A} $\forall x: \text{times}(\text{inv}(x), x) = e,$
- \textcircled{A} $\forall x, y, z: \text{times}(x, \text{times}(y, z)) = \text{times}(\text{times}(x, y), z).$

Here, x , y and z are variables, while e is a symbolic constant. The functions $\text{times}(\cdot)$ and $\text{inv}(\cdot)$ are user-defined, uninterpreted functions that are implicitly declared. Now, we may want to have Simplify prove the proposition

$$\forall x, y: \text{times}(\text{times}(\text{inv}(y), \text{inv}(x)), \text{times}(x, y)) = e,$$

which indeed succeeds.

Next, let us define a predicate $\text{isInverse}(x, y)$ holding true iff x is an inverse of y :

$$\textcircled{A} \quad \forall x, y: (\text{isInverse}(x, y) \Leftrightarrow \text{times}(x, y) = e).$$

According to the discussion in the previous section, in reality we cannot define the new predicate directly, but would have to use a quasi-predicate instead. We may now rephrase the previous proposition as

$$\forall x, y: \text{isInverse}(\text{times}(\text{inv}(y), \text{inv}(x)), \text{times}(x, y)).$$

Of course, this proposition is also successfully proven by Simplify.

2.3. Quantifiers and Triggering Patterns

An important aspect is Simplify's handling of quantifications. If an existentially quantified predicate is postulated to be true, Simplify eliminates the existential quantifier using the well-known technique of Skolemization. On the other hand, if a universally

quantified predicate is postulated to be true, Simplify produces a *matching rule* with some *triggering patterns*.

A matching rule is an internal representation of a universally quantified predicate in a form that enables the prover to produce potentially relevant instantiations of its body in response to the detection of ground terms matching the triggering patterns. For instance, postulating the axiom

$$\textcircled{A} \quad \forall x: \text{times}(e, x) = x$$

produces a matching rule with the triggering pattern $\text{times}(e, x)$. Whenever the prover finds a ground term of the form $\text{times}(e, x_0)$, it will instantiate the body of the axiom with the substitution $x := x_0$, i.e., it will postulate $\text{times}(e, x_0) = x_0$.

Generally, a triggering pattern for a universal quantification has to be a single term in the body of the quantification that contains all quantified variables. Moreover, it must not be a single variable².

For the rest of this document, we use underlining to emphasize the triggering patterns for universal quantifications postulated to hold true. For example, we would write the group theory identity axiom as

$$\textcircled{A} \quad \forall x: \underline{\text{times}(e, x)} = x.$$

Multi-Triggers

Sometimes we are obliged to use a *set* of terms as a triggering pattern instead of a single term. For instance, for a quantified predicate like

$$\forall s, t, x: (\text{member}(x, s) \wedge \text{subset}(s, t) \Rightarrow \text{member}(x, t))$$

no single term is an adequate trigger, as no single term contains all the quantified variables. An appropriate triggering pattern is the set of terms $\{\text{member}(x, s), \text{subset}(s, t)\}$:

$$\forall s, t, x: (\underline{\text{member}(x, s)} \wedge \underline{\text{subset}(s, t)} \Rightarrow \text{member}(x, t)).$$

Such a triggering pattern that contains more than one term is called a *multi-trigger*, and the terms are called its *constituents*. The constituents of a multi-trigger have to collectively contain all quantified variables and a constituent itself may not be a single variable. Although sometimes needed, multi-trigger matching is generally more expensive than single-trigger matching.

²There are several more restrictions as to what kind of terms may act as triggering patterns. However, they are not relevant here.

2.3. Quantifiers and Triggering Patterns

Note that triggering patterns are sets of *terms*, not predicates. Thus, it is not possible to specify the following trigger:

$$\forall s, t, x: \underline{\text{member}(x, s) \wedge \text{subset}(s, t)} \Rightarrow \text{member}(x, t).$$

Choosing Triggering Patterns

The choice of triggering patterns for universal quantifications can impact both the completeness and the performance of Simplify. If too liberal a trigger is chosen, the prover can be swamped with irrelevant instances. If too conservative a trigger is chosen, an instance crucial to a proof might be excluded. For an example of the effect of trigger selection, consider [4].

Simplify has heuristics for automatically choosing triggering patterns, but allows a user to override the heuristics and specify the triggering patterns explicitly.

3. Data and State Model Formalization

The data and state model of a programming language describes its types, values, objects and object states. Among others, it is used as a formal foundation for the axiomatic semantics of the programming language and to make properties of programs such as type information available for specifications and proofs.

A formalization of the data and state model of the target language of Jive is particularly needed for Isabelle to be able to prove the general verification conditions (lemmas) that are generated during a Jive proof. This way, the data and state model acts as interface between Jive and Isabelle.

The data and state model used for the target language of Jive is based on the model described by Müller and Poetzsch-Heffter [8]. It is formalized in program independent and program dependent Isabelle theories. The program independent part formalizes, e.g., types and values, while the program dependent part contains information about the types, fields, etc. declared in a program.

To use Simplify as an alternative theorem prover in Jive, a Simplify formalization of the data and state model of the target language of Jive needs to be available. This chapter discusses the translation of the data and state model formalization from Isabelle to Simplify.

3.1. Typed vs. Untyped Logics

The major obstacle in translating the formal data and state model from Isabelle to Simplify is Simplify's type freeness, whereas the Isabelle formalization makes heavy use of custom data types. Consequently, all type information in the Isabelle formalization needs to be encoded into the logic of Simplify.

To demonstrate possible ways to achieve such a transformation, let us walk through an illustrative example that is representative for all major issues that arise in the actual translation of the formal data and state model. We utilize some conventional Isabelle-like syntax for the specification of the typed formalization and assume we have (i) a data type S with three constants (or, equivalently, nullary constructors) S_1 , S_2 and S_3 ,

(ii) a data type T with a constant T_1 and a constructor T_2 taking a value of type S as argument, and finally (iii) a data type U with a constructor U_1 taking a value of the built-in integer type **int** as argument:

$$\begin{aligned} \text{datatype } S &= S_1 \mid S_2 \mid S_3, \\ \text{datatype } T &= T_1 \mid T_2(S), \\ \text{datatype } U &= U_1(\text{int}). \end{aligned} \tag{3.1}$$

Further, let $f: T \times U \rightarrow S$ be a function defined by a case distinction over the possible constructors of T , independently of the second argument of type U :

$$f(t, u) := \begin{cases} S_1 & , \text{ if } t = T_1; \\ S_2 & , \text{ if } t = T_2(s). \end{cases} \tag{3.2}$$

Finally, we have three (valid) propositions we would like to prove:

$$\forall(t :: T, u :: U): f(t, u) \neq S_3, \tag{3.3}$$

$$f(T_2(S_1), U_1(42)) = S_2, \tag{3.4}$$

$$(\forall(u :: U): Q(u)) \Rightarrow Q(U_1(84)). \tag{3.5}$$

Proposition (3.3) is a general statement about the function f , where the variables t and u are quantified over all values of types T and U , respectively. In contrast, proposition (3.4) applies the function f to some constants without involving quantifiers. Proposition (3.5) consists of an assumption about some predicate Q over U and a simple instantiation thereof.

To support the claimed relation to reality, we remark that type declarations such as (3.1) can be found throughout the Isabelle formalization of the data and state model. In particular, data type U resembles the data type `Value` in the Isabelle formalization, where U_1 represents a constructor for integer values. During verification of actual programs, formulas similar to proposition (3.3) arise in proof obligations resulting from a `modifies` clause¹, while verification conditions similar to proposition (3.4) are generated when reasoning about object store accesses and updates. Lastly, implications having a universal quantification in the antecedent, such as (3.5), emerge, e.g., in the verification of constructor methods, pure methods and methods involving calls to other methods.

3.1.1. Straight Forward Approach

The straight forward approach to encode the type declarations (3.1) in the Simplify logic would be to introduce a predicate `isX` for every type X that characterizes the values of

¹An *modifies clause* (also called *assignable clause*) is used to explicitly state what object store locations may be modified during a method execution. Typically, the resulting proof obligation involves a universal quantification over all locations.

type X :

$$\textcircled{A} \quad \forall x: (\underline{\text{isS}(x)} \Leftrightarrow x = S_1 \vee x = S_2 \vee x = S_3), \quad (3.6)$$

$$\textcircled{A} \quad \forall x: (\underline{\text{isT}(x)} \Leftrightarrow x = T_1 \vee \exists y: (x = T_2(y) \wedge \text{isS}(y))), \quad (3.7)$$

$$\textcircled{A} \quad \forall x: (\underline{\text{isU}(x)} \Leftrightarrow \exists y: (x = U_1(y) \wedge \text{isInt}(y))), \quad (3.8)$$

$$\textcircled{A} \quad \forall x: (\underline{\text{isInt}(x)} \Leftrightarrow \mathbf{True}). \quad (3.9)$$

Note the necessary existential quantifier for constructors with at least one argument. Furthermore, observe that the underlined triggering patterns are the only possible ones for these axioms (cf. Section 2.3). As another remark, we note that the type **int** is mapped directly to Simplify's individual values, since it would obviously be impossible to enumerate all integer values in the same way as for the `isS` predicate. So, in actual fact, `isInt(x)` does not restrict x in any way and the `isInt(y)` conjunct in (3.8) could equivalently be dropped. We shall come back to this issue later in this section.

The type declarations (3.1) also entail a number of implicit type laws (cf. Isabelle tutorial [3], Sections 2.4.2 and 8.5.2). In particular, we add the following axioms about the distinctness and injectivity of the constructors:

$$\textcircled{A} \quad S_1 \neq S_2 \wedge S_1 \neq S_3 \wedge S_2 \neq S_3 \quad (\text{distinctness of the } S\text{-constructors}),$$

$$\textcircled{A} \quad \forall x: T_1 \neq T_2(x) \quad (\text{distinctness of the } T\text{-constructors}),$$

$$\textcircled{A} \quad \forall x, y: (\underline{T_2(x)} = \underline{T_2(y)} \Rightarrow x = y) \quad (\text{injectivity of } T_2),$$

$$\textcircled{A} \quad \forall x, y: (\underline{U_1(x)} = \underline{U_1(y)} \Rightarrow x = y) \quad (\text{injectivity of } U_1).$$

We can now use the declared `isX` predicates to fix the types of variables occurring in formulas. So, for the definition (3.2) of function f we introduce the following axioms:

$$\textcircled{A} \quad \forall u: (\underline{\text{isU}(u)} \Rightarrow \underline{f(T_1, u)} = S_1), \quad (3.10)$$

$$\textcircled{A} \quad \forall s, u: (\underline{\text{isS}(s)} \wedge \underline{\text{isU}(u)} \Rightarrow \underline{f(T_2(s), u)} = S_2). \quad (3.11)$$

The choice of the triggering patterns for these axioms is discussed further below. In a similar way, the encodings of propositions (3.3) to (3.5) look as follows:

$$\forall t, u: (\text{isT}(t) \wedge \text{isU}(u) \Rightarrow f(t, u) \neq S_3), \quad (3.12)$$

$$f(T_2(S_1), U_1(42)) = S_2, \quad (3.13)$$

$$(\forall u: (\text{isU}(u) \Rightarrow Q(u))) \Rightarrow Q(U_1(84)). \quad (3.14)$$

This way, proposition (3.12) can successfully be proven in Simplify, since the `isT(t)` and `isU(u)` predicates restrict the values of t and u to their respective types and Simplify

can perform a case distinction between these values. However, Simplify is unable to prove propositions (3.13) and (3.14), because, when encountering the constant S_1 , say, Simplify cannot derive $\text{isS}(S_1)$ (which is necessary to apply axiom (3.11)), since the triggering pattern in definition (3.6) of isS is $\text{isS}(x)$ itself. Consequently, we have to introduce additional helper axioms (which are actually provable by Simplify) such as:

$$\textcircled{A} \quad \text{isS}(S_1) \wedge \text{isS}(S_2) \wedge \text{isS}(S_3), \quad (3.15)$$

$$\textcircled{A} \quad \text{isT}(T_1) \wedge \forall s: (\text{isS}(s) \Rightarrow \text{isT}(T_2(s))), \quad (3.16)$$

$$\textcircled{A} \quad \forall x: (\text{isInt}(x) \Rightarrow \text{isU}(U_1(x))). \quad (3.17)$$

Note that (3.17) essentially reads

$$\textcircled{A} \quad \forall x: \text{isU}(U_1(x)). \quad (3.17a)$$

With the addition of these helper axioms, Simplify manages to successfully prove propositions (3.13) and (3.14).

While the described approach works perfectly well in theory, it has two major drawbacks in practical use with Simplify:

Inefficient triggering patterns. What should be the triggering patterns for the axioms (3.10), (3.11), (3.16) and (3.17), respectively? If we choose $f(T_1, u)$ as the triggering pattern for axiom (3.10), then Simplify is likely to instantiate the axiom with a substitution $u := u_0$ even where u_0 is not known to satisfy $\text{isU}(u_0)$, which may cause the prover to do useless case splits. To reduce the likelihood of producing such futile instantiations of the axiom, we can use the terms $\text{isU}(u)$ and $f(T_1, u)$ together as a multi-trigger. However, the disadvantage of this approach is the inefficiency of multi-triggers compared to single-term triggering patterns. A similar discussion goes for (3.11), (3.16) and (3.17).

Inefficiency due to unbounded existentially quantified variables. Whenever Simplify encounters a term $\text{isU}(x)$, such as in (3.12) and (3.17a), making use of (3.8), it may deduce $x = U_1(y_0)$ with an arbitrary instantiation y_0 of y , since the existentially quantified variable y is in fact not bounded at all. As a result, especially with larger formalizations and more complex propositions that involve instantiations of more than one single axiom, Simplify is prone to getting bogged down by a lot of pointless instantiations. This slows down Simplify substantially, in a way that is not acceptable.

3.1.2. The ‘as’ Trick

This subsection introduces a simple a way to overcome the inefficient trigger problem described in the straight forward approach: the so-called ‘*as*’ trick [7].

For every type X , we introduce two additional axioms describing a function asX :

$$\begin{aligned} \textcircled{A} \quad \forall x: (\text{isX}(x) \Rightarrow \text{asX}(x) = x), \\ \textcircled{A} \quad \forall x: \text{isX}(\text{asX}(x)). \end{aligned} \tag{3.18}$$

Intuitively, asX can be seen as a function that casts any value into a value of type X , while being the identity function on values that are already of type X . With help of the axioms (3.18), the following equivalences can easily be shown to hold for an arbitrary predicate P :

$$\begin{aligned} \forall x: (\text{isX}(x) \Rightarrow P(x)) &\equiv \forall x: P(\text{asX}(x)), \\ \exists x: (\text{isX}(x) \wedge P(x)) &\equiv \exists x: P(\text{asX}(x)). \end{aligned}$$

In other words, this allows us to get rid of the bothering $\text{isX}(x)$ terms in axioms and propositions and instead use $\text{asX}(x)$ as part of a single-term triggering pattern, which should be efficient to match. For instance, we can reformulate axiom (3.11) as

$$\textcircled{A} \quad \forall s, u: \underline{f(T_2(\text{asS}(s)), \text{asU}(u))} = S_2,$$

while maintaining equivalence.

Though employing the ‘as’ trick solves the problem of the inefficient multi-triggers of the straight forward approach, the efficiency issue caused by unbounded existentially quantified variables still persists.

3.1.3. No Type Information at All

Another option to transform the typed Isabelle formalization to Simplify is to simply throw away all type information. So, the isX and asX predicates as well as the helper axioms (3.15) to (3.17) are not necessary anymore. The function definition (3.2) is then encoded as

$$\begin{aligned} \textcircled{A} \quad \forall u: \underline{f(T_1, u)} = S_1, \\ \textcircled{A} \quad \forall s, u: \underline{f(T_2(s), u)} = S_2, \end{aligned}$$

and the transformations of propositions (3.3) to (3.5) look like

$$\forall t, u: f(t, u) \neq S_3, \tag{3.19}$$

$$f(T_2(S_1), U_1(42)) = S_2, \tag{3.20}$$

$$(\forall u: Q(u)) \Rightarrow Q(U_1(84)). \tag{3.21}$$

3.1. Typed vs. Untyped Logics

Removing the type information makes partial functions total, as their arguments are not restricted to values of their respective types anymore.

On the plus side, this formalization is very efficient, since there are neither slow multi-triggers nor unbounded existentially quantified variables. Propositions (3.20) and (3.21) can easily be proven by Simplify. Nevertheless, there are two disadvantages: Firstly, proposition (3.19) is invalid and therefore cannot be proven, since the quantified variables t and u are not restricted to the values of their respective types. Secondly, the removal of the type information is a potential source of unsoundness. E.g., in this setting, we are able to prove something like

$$f(T_2(0), S_1) = S_2, \quad (3.22)$$

which is nonsensical since the ground term $f(T_2(0), S_1)$ is ill-typed. As another example, the valid formula

$$\forall(x :: S) : (x = S_1 \vee x = S_2 \vee x = S_3) \quad (3.23)$$

would yield

$$\forall x : (x = S_1 \vee x = S_2 \vee x = S_3) \quad (3.24)$$

after dropping the type information, which may clearly lead to unsound results when postulated to be true. Note, however, that for formulas of form (3.24) describing a case split over the values of a certain type, there exists no valid triggering pattern for the quantification body (since a triggering pattern must be a term but not a single variable) and Simplify issues a corresponding warning. In other words, Simplify would not be able to ever instantiate the body of the quantification, anyway. So, removing the type information in formulas of such a type can in fact cause no unsoundness in our setting.

To see an example of introduced unsoundness with formulas that *have* a valid Simplify triggering pattern consider the following, somewhat contrived singleton data type H and function $h: H \rightarrow H$:

datatype $H = H_1$
 $h(H_1) := H_1.$

We further assume the following two valid formulas:

$$\forall(x :: H) : h(x) = H_1, \quad (3.25)$$

$$\forall(x :: H, y :: H) : h(x) = h(y) \Rightarrow x = y. \quad (3.26)$$

Dropping the type information of (3.25) and (3.26) would lead to the formulas

$$\forall x : \underline{h(x)} = H_1, \quad (3.27)$$

$$\forall x, y : \underline{h(x)} = \underline{h(y)} \Rightarrow x = y, \quad (3.28)$$

both of which have a valid triggering pattern, as indicated by underlining. The bodies of (3.27) and (3.28) could then be instantiated with the integers 1 and 2 to obtain $1 = 2$, which illustrates the potential introduction of unsoundness.

We will have a closer look at this unsoundness issue in the next subsection.

3.1.4. Approach Taken

In the actual taken approach we cherry-pick from both the ‘as’ trick approach and the approach discarding all type information. This way, we end up with a formalization that allows for efficient yet flexible proofs. Intuitively, we start off with the untyped approach and selectively “add back” the type information where it is needed. Recall that the two major disadvantages of completely removing the type information are (i) potential introduction of unsoundness and (ii) potential reduction in completeness, in particular the fact that proposition (3.19) fails to be proven.

Introduction of Unsoundness

We justify in an informal argument that in our setting no unsoundness can actually be introduced by removing the type information.

To begin with, it is important to know that every new verification condition generated by Jive is properly type checked. In particular, this includes lemmas that involve user input. Since the Isabelle formalization itself is clearly well-typed, we can be sure that ill-typed formulas such as (3.22) will never be sent to Simplify, neither as background predicate, nor as verification condition to be proven.

Moreover, all background axioms and lemmas of the data and state model formalization have been examined so as to make sure that removing the type information cannot cause any unsoundness. In other words, the formalization contains no such formula as (3.23), (3.25) or (3.26), where dropping the types may introduce unsoundness.

In addition, removing the type information in generated verification conditions that do not involve user input is safe as well, since the generated universal quantifications do not rely on the fact that the quantified variables are of a certain type.

Remain the verification conditions with user input. Any typical user input in practical program verification (e.g., a loop invariant) does not contain critical formulas as (3.23), (3.25) or (3.26) either. So, removing the type information in this case cannot cause unsoundness either. However, it should be noted that a malicious user might be able to cause unsoundness by entering a formula which is

- postulated to be true (e.g., on the left-hand side of an implication),

- has a valid Simplify triggering pattern,
- is valid in the typed world and
- is invalid when the type information is removed,

although the author has no concrete example of such an input (a user is not allowed to introduce new types).

In all, this completes the informal argument that no unsoundness can be introduced by discarding the type information in a formula in any practical setting. As future work, this argument may be backed up by a formal proof.

Reduction in Completeness

Let us now turn to the incompleteness issue or, more specifically, the failure to prove proposition (3.19). In order to remedy this, we want to preserve the type information for propositions of such a form, so that the values of the quantified variables can be restricted to their respective types. It might appear tempting to generally preserve the type information in *every* universal quantification. However, while this would allow a successful proof of proposition (3.19), proving proposition (3.21) would then fail in turn, since the quantification appears in the antecedent of the implication. Though introducing the helper axioms (3.15) to (3.17) would again solve this new problem, in order to keep the formalization compact and to avoid critical isU predicates, we want to do without the helper axioms. Besides, we only want to add type information where it is really necessary.

So, instead, the following heuristic is used to handle the type information in quantifications:

if the quantification is universal and either appears in the consequent of a top-level implication or is a top-level predicate itself **then**
 preserve the type information of the quantified variables;
else
 remove the type information of the quantified variables.

Using this strategy, propositions (3.3) to (3.5) are translated as

$$\begin{aligned} \forall t, u: f(\text{asT}(t), \text{asU}(u)) &\neq S_3, \\ f(T_2(S_1), U_1(42)) &= S_2, \\ (\forall u: P(u)) &\Rightarrow P(U_1(84)), \end{aligned}$$

all of which are successfully proven in Simplify (note the preserved type information in the first proposition). We note that in order to preserve the type information for certain

quantifications we need to keep the definitions of the `isX` and `asX` predicates, respectively, for each type X . This means in particular that the existential quantifiers that appear in `isX` predicate definitions are still present in the final formalization. However, since `asX` expressions appear only in verification conditions where the type information is preserved (and not in *every* formula of the formalization as would be the case in a completely typed setting), the existential quantifiers are only triggered for a little number of variables, which does not cause Simplify to dramatically slow down.

The above described procedure is implemented in the Simplify unparser (cf. Chapter 4). We further remark that preserving type information obviously does not affect soundness in any way.

In summary, the presented approach combines the flexibility of the ‘as’ trick approach with the efficiency of the approach removing all type information. Table 3.1 recapitulates the pros and cons of all discussed approaches to encoding type information in an untyped logic.

3.2. Translation

Once the basic issue of typed vs. untyped is solved, the translation of the formal data and state model from Isabelle to Simplify proceeds in a pretty straight forward way. The program independent part of the data and state model is formalized in 7 files lying in the directory `theories/Simplify/` of the Jive distribution. The program dependent part consists of 4 files that are generated in a subdirectory prefixed with `Simplify_`, relative to the directory of the program source file.

In the following, we mention some noteworthy points concerning the translation, including deviations from the Isabelle formalization. The squared brackets indicate the corresponding file name.

Proven Lemmas vs. Axioms

The Isabelle formalization is sound in that all occurring lemmas are actually proven in Isabelle itself. What is more, there is a concrete implementation of the object store that is proven to fulfill the required axioms of the axiomatic description in Müller’s and Poetzsch-Heffter’s data and state model.

In contrast, the final Simplify formalization consists merely of axioms. However, many of those are helper axioms that are in fact provable by Simplify. During the development of the Simplify formalization, axioms were avoided and replaced by proven lemmas wherever it was possible in order to reduce the possibility of accidentally introducing

unsoundness. In the end, the proven lemmas were then turned into axioms, so that Simplify does not have to prove them over and over again.

Subtype Relation

In the Isabelle formalization, the subtype relation is defined to be the transitive closure of the direct subtype relation and proven to form a partial order. As opposed to that, in the Simplify formalization, the transitive closure is explicitly calculated and all pairs in subtype relation are enumerated as axioms [`SubtypeDep.simp`]. This avoids recursive dependencies not suited for an automatic theorem prover such as Simplify, which is not equipped for induction.

Spurious Model Locations

In the current Isabelle formalization there is no distinction between concrete field identifiers and model field identifiers. Therefore, it is possible to create a concrete location with a model field identifier or, conversely, a model location with a concrete field identifier. However, model locations constructed with a concrete field are completely unspecified, which causes proofs of modifies clauses to fail.

As a quick fix for the Simplify integration, representation functions that always yield zero-equivalent values have been added for spurious model locations constructed with concrete fields [`UnivSpec.simp`]. However, in the long run it is preferable to develop a cleaner solution (for both the Isabelle and Simplify formalizations) that distinguishes between model and concrete field identifiers. As soon as such a change is implemented, the aforesaid representation functions may be removed.

Arrays and Static Fields

Both the Isabelle and Simplify formalizations handle arrays and static fields, although such constructs are not supported in Java-KEx, the current target language of Jive. This way, both models are ready for future extensions of Jive.

	Typed World	Straight Forward Approach	‘as’ Trick	Discarding Types	Approach Taken
Type Declaration Ⓐ	datatype $S = S_1 \mid S_2 \mid S_3$ datatype $T = T_1 \mid T_2(S)$ datatype $U = U_1(\mathbf{int})$	$\forall x: (\mathbf{isS}(x) \Leftrightarrow x = S_1 \vee x = S_2 \vee x = S_3)$ $\forall x: (\mathbf{isT}(x) \Leftrightarrow x = T_1 \vee \exists y: (x = T_2(y) \wedge \mathbf{isS}(y)))$ $\forall x: (\mathbf{isU}(x) \Leftrightarrow \exists y: (x = U_1(y) \wedge \mathbf{isInt}(y)))$ $\forall x: (\mathbf{isInt}(x) \Leftrightarrow \mathbf{True})$ (plus distinctness and injectivity) (plus helper axioms)	as in straight forward approach plus (for each type X): $\forall x: (\mathbf{isX}(x) \Rightarrow \mathbf{asX}(x) = x)$ $\forall x: \mathbf{isX}(\mathbf{asX}(x))$	only distinctness and injectivity	as in ‘as’ trick (minus helper axioms)
Function Declaration Ⓐ	$f: T \times U \rightarrow S$ $f(t, u) := \begin{cases} S_1, & \text{if } t = T_1 \\ S_2, & \text{if } t = T_2(s) \end{cases}$	$\forall u: (\mathbf{isU}(u) \Rightarrow f(T_1, u) = S_1)$ $\forall s, u: (\mathbf{isS}(s) \wedge \mathbf{isU}(u) \Rightarrow f(T_2(s), u) = S_2)$	$\forall u: f(T_1, \mathbf{asU}(u)) = S_1$ $\forall s, u: f(T_2(\mathbf{asS}(s)), \mathbf{asU}(u)) = S_2$	$\forall u: f(T_1, u) = S_1$ $\forall s, u: f(T_2(s), u) = S_2$	as in discarding types
Prop. 1	$\forall (t :: T, u :: U): f(t, u) \neq S_3$	$\forall t, u: (\mathbf{isT}(t) \wedge \mathbf{isU}(u) \Rightarrow f(t, u) \neq S_3)$	$\forall t, u: f(\mathbf{asT}(t), \mathbf{asU}(u)) \neq S_3$	$\forall t, u: f(t, u) \neq S_3$	as in ‘as’ trick
Prop. 2	$f(T_2(S_1), U_1(42)) = S_2$	$f(T_2(S_1), U_1(42)) = S_2$	$f(T_2(S_1), U_1(42)) = S_2$	$f(T_2(S_1), U_1(42)) = S_2$	as in discarding types
Prop. 3	$(\forall (u :: U): Q(u)) \Rightarrow Q(U_1(84))$	$(\forall u: (\mathbf{isU}(u) \Rightarrow Q(u))) \Rightarrow Q(U_1(84))$	$(\forall u: Q(\mathbf{asU}(u))) \Rightarrow Q(U_1(84))$	$(\forall u: Q(u)) \Rightarrow Q(U_1(84))$	as in discarding types
Comments		⊕ all propositions provable ⊕ soundness ⊖ inefficient triggers ⊖ inefficiency due to unbounded existentially quantified variables	⊕ all propositions provable ⊕ soundness ⊕ efficient triggers ⊖ inefficiency due to unbounded existentially quantified variables	⊕ efficiency ⊖ proposition 1 not provable ⊖ potential introduction of unsoundness	⊕ all propositions provable ⊖ soundness (no formal proof) ⊕ efficiency

Table 3.1: Encoding type information in an untyped logic

Miscellaneous

- Some additional (provable) helper axioms have been added to improve Simplify's performance [`Value.simp`, `Location.simp`]. An example of such an axiom is

$$\begin{aligned} \forall (v :: \text{Value}, C :: \text{CTypeId}): \\ (\text{typeof}(v) <: \text{CClassT}(C) \wedge v \neq \text{nullV} \wedge C \neq \text{java.lang.Object}) \\ \Rightarrow \text{isObjV}(v), \end{aligned}$$

which is to say that if a value is a subtype of a certain class and not null, then it is an object value. The additional precondition $C \neq \text{java.lang.Object}$ is needed to exclude array values, which are not considered object values in the formalization. (A variant of the above helper axiom for the case $\text{typeof}(v) = \text{CClassT}(\text{java.lang.Object})$ is also present in the formalization.)

- As Simplify lacks built-in integer division and modulo functions, some axiomatic descriptions for them, taken from the ESC/Java source code, have been added [`UnivSpecIndep.simp`].
- The Isabelle theory file `StoreProperties.thy` has been discarded in the translation of the formalization, as it mostly contains recursive reachability properties that are difficult to handle for an automatic theorem prover and hardly ever used in proofs anyway.

4. Simplify Unparser

The Simplify unparser is a module that translates formulas from Jive’s internal representation to a string containing the formula in the Simplify input syntax. This is needed for the generation of the program dependent part of the data and state model (more specifically, to emit type and program invariants as well as access functions of model locations) and, obviously, to unparse the verification conditions when they are sent to Simplify. This chapter deals with the implementation of such a module.

4.1. Terms and Formulas

Jive uses a handy tool called *Katja* [10] to describe its internal representation of terms and formulas. *Katja* takes specification files with definitions of order-sorted data types as input and automatically generates the corresponding Java classes. Relevant excerpts of the current *Katja* specification files for Jive terms and formulas are given in Appendix A.2.

Comparing Jive’s and Simplify’s model of terms and formulas (cf. Section 2.1 and Appendix A.1), there is an inherent mismatch. Most notably, in Jive, truth values are considered terms, which can in turn be promoted to formulas using the `isTrue` constructor. Hence, there are boolean and relational operators that live completely in the term space, i.e., they take terms as arguments and are terms themselves. This design is motivated by the desire to map Java expressions to terms. In contrast, in Simplify, boolean connectivities can only be applied to formulas, and relational operations are considered formulas, not terms.

For instance, take the Jive formula

```

isTrue(
  BinaryOpTerm(
    BinaryOpTerm(
      BinaryOpTerm(
        IntegerLiteral(1),
        TLessEq(),
        IntegerLiteral(2)
      ),
      TEq(),
      ProgVar(x),
    ),
    TLogicalAnd(),
    ProgVar(y)
  )
)

```

(4.1)

which might have resulted from the Java expression

$$((1 \leq 2) == x) \& y,$$

where x and y are program variables of type `Boolean`. If we naively unparsed this formula as

$$((1 \leq 2) = x) \wedge y,$$

we would end up with errors because, in the Simplify world, $(1 \leq 2)$ is a formula used where a term is expected and, conversely, y is a term used where a formula is expected.

To overcome the formula/term discrepancy between Jive and Simplify, we introduce additional Simplify axioms describing term space operators that are missing in Simplify, making use of Simplify's special term constant `@true` that we already encountered in Section 2.1. For instance, to introduce a term space 'logical and' operator we add the axiom

$$\textcircled{A} \quad \forall x, y: (\text{TLogicalAnd}(x, y) = \text{@true} \Leftrightarrow x = \text{@true} \wedge y = \text{@true})$$

and similar ones for other boolean term space operators. Note the distinction between the introduced term space operator `TLogicalAnd(., .)` and the built-in predicate space operator " \wedge ". Likewise, to introduce a term space 'less or equal' operator we add the axiom

$$\textcircled{A} \quad \forall x, y: (\text{TLessEq}(x, y) = \text{@true} \Leftrightarrow x \leq y)$$

and similar ones for other relational term space operators. All these axioms can be found in the file `UnivSpecIndep.simp`.

In all, this leads to a one-to-one correspondence between the term space operators in Jive and Simplify. Finally, the formula `isTrue(t)` is translated as `t = @true`. Summing up, the unparsing of our example Formula (4.1) looks like this:

$$\text{TLogicalAnd}(\text{TEq}(\text{TLessEq}(1, 2), x), y) = \text{@true}.$$

As an aside, we remark that we treat the ‘conditional and’ and ‘conditional or’ operators like ‘logical and’ and ‘logical or’ operators, respectively, which is the same way they are handled in the Isabelle unparser. By virtue of the semantics of JML, this is fine in case the operators are used in specifications.

4.2. Implementation

Having smoothed out the differences between the term/formula design of Jive and Simplify, the Simplify unparser can be implemented as a simple, straight forward visitor of a Jive formula. This is done in the class `SimplifyUnparser`¹.

However, there are still some constructs that need some special treatment, of which the most important ones are mentioned in the following.

Set Terms

Set terms such as $\{t_1, \dots, t_k\}$ are not directly supported in Simplify. Nonetheless, expressions of the form

$$M = \{t_1, \dots, t_k\}, \quad (4.2)$$

where M is a logical variable, can easily be mapped to

$$\forall x: (M(x) = \text{@true} \Leftrightarrow x = t_1 \vee \dots \vee x = t_k), \quad (4.3)$$

where M acts as a unary quasi-predicate symbol.

Such a translation works only if (i) the set expression is of the form of (4.2) (or the symmetric version) and (ii) the expression appears as top-level term, i.e., as direct child of `isTrue`. The reason for condition (ii) is the involved universal quantification, which cannot act as a term. In other words, constructs such as $(M = \{a, b\}) = (1 \leq 2)$ or

¹Due to some Java 1.4 and 1.5 incompatibility issues, some parts of the current Jive code need to be duplicated. This includes the Simplify unparser, and its two instances can be found in the packages `jive.inout.formula.simplify` and `jive.frontend.tpunparsers.simplify`.

$\{a, b\} = \{1, 2\}$ are not supported by the momentary version of the Simplify unparser. This is justified by the fact that set terms are currently only used in the form of (4.2), where they represent the modifiable locations mentioned in a modifies clause.

In accordance with (4.3), set membership relations such as $t \in M$ are then unparsed as

$$M(t) = \text{@true}.$$

Conditional Terms

Since the conditional term

$$\text{if } e_b \text{ then } e_1 \text{ else } e_2$$

(corresponding to the Java expression `eb ? e1 : e2`) has no equivalent construct in Simplify, we add the axioms (also in the file `UnivSpecIndep.simp`)

- Ⓐ $\forall e_b, e_1, e_2: (e_b = \text{@true} \Rightarrow \text{ConditionalTerm}(e_b, e_1, e_2) = e_1),$
- Ⓐ $\forall e_b, e_1, e_2: (e_b \neq \text{@true} \Rightarrow \text{ConditionalTerm}(e_b, e_1, e_2) = e_2)$

and simply unparse it as

$$\text{ConditionalTerm}(e_b, e_1, e_2).$$

False (In)equalities

The term constant `True()` in Jive can directly be mapped to the corresponding constant `@true` in Simplify. But how should we unparse the Jive term constant `False()`? First of all, we introduce a corresponding term constant `@false` for Simplify by the following axiom:

$$\text{Ⓐ } \text{@false} \neq \text{@true}.$$

While this axiom allows us to prove

$$x_0 = \text{@false} \Rightarrow x_0 \neq \text{@true}$$

for a constant x_0 , it is not sufficient to prove the converse

$$x_0 \neq \text{@false} \Rightarrow x_0 = \text{@true}, \tag{4.4}$$

when x_0 is not restricted to boolean values. This poses a problem, since propositions of the form of (4.4) may arise in practice (after the type information about x_0 has been removed). As discussed in Subsection 3.1.4, we could introduce an axiom such as

$$\text{Ⓐ } \forall x: (x = \text{@true} \vee x = \text{@false}),$$

without causing unsoundness. However, this would not work practically, since the quantification body does not have a valid triggering pattern to choose for Simplify, and Simplify issues a corresponding warning.

Instead, we take the following approach: Whenever the constant `@false` appears in an equality or inequality, we eliminate it by translating the (in)equality to an equivalent (in)equality containing the constant `@true`. For instance, we write $x \neq \text{@true}$ instead of $x = \text{@false}$, and $x = \text{@true}$ instead of $x \neq \text{@false}$. Using this strategy, (4.4) is unparsed as

$$x_0 = \text{@true} \Rightarrow x_0 = \text{@true},$$

which is trivial to prove.

Types and Binding Formulas

As discussed in Subsection 3.1.4, all type information about variables and constants is discarded, with the exception of universal quantifications that either appear in the consequent of a top-level implication or are top-level predicates themselves. A quantified variable x of type X in such a quantification is unparsed to `asX(x)`, preserving x 's type information using the 'as' trick.

Avoiding Name Clashes

There is a potential risk of name clashes between program variables, logical variables introduced during Jive proof sessions and declared constants (including functions) used in the formal data and state model. In the Isabelle unparser, this is solved by suffixing every logical variable with an apostrophe (`'`), which is not allowed to be used in Java program variable names. What is more, there can be no name clashes between program variables and declared constants, since every use of a declared constant has to be qualified with the declaring Isabelle theory, which involves a dot (`.`).

However, there is no such a thing as a theory hierarchy in Simplify, and all declared constants lie in the same "namespace". So, there could potentially be name clashes between program variables and declared constants (e.g., a program variable named `update` would coincide with the store update function of the same name). To avoid such cases, the Simplify unparser suffixes every program variable with a percent sign (`%`), of which we are sure that it is not used in logical variables or declared constants.

5. Workflow and GUI Integration

This chapter addresses the high-level integration of Simplify into the Jive verification workflow and GUI.

Program proofs in Jive are represented by trees that are built from instances of the axioms and rules of the underlying programming logic. General lemmas (or verification conditions) to be proven in a general theorem prover (such as Simplify or Isabelle) result from applying special kinds of rules. A lemma is always attached to the proof tree node that documents this rule application.

Jive provides two user-interfaces to program proofs, the *tree view* and the *text view*. The tree view is a graphical representation of the proof tree showing all details of the program proof, whereas the text view comprises a more compact view that enables an embedding of program proof information into the program text. Since the text view was not in a working state during this project's duration, Simplify has only been integrated into the tree view, leaving the integration into the text view open for some future work.

5.1. Interaction with Simplify

Every lemma has an associated *Simplify state* whose value range is:

- **initial**: the lemma has not been sent to Simplify
- **pending**: the lemma has been sent to Simplify, but no result is available yet
- **proven**: the lemma has been successfully proven by Simplify
- **disproven**: the lemma has been disproven by Simplify
- **timed_out**: the Simplify proof of the lemma has exceeded the time/iteration limit (cf. Section 5.2)

In the GUI, the Simplify state of a lemma is displayed next to the “LEMMA” label just above the actual lemma (cf. Figure 5.4). To have a transparent integration for users not willing to use Simplify, the lemma state **initial** is not explicitly shown.

Since a lemma does not affect any part of the program at hand (after all, this is why they are exported to a general theorem prover), there is no reason the user should be blocked while waiting for a Simplify proof. Hence, the interaction with Simplify happens in an asynchronous way. In other words, after a lemma has been sent to Simplify, the user is able to immediately continue his/her proving work (perhaps at another location in the proof tree) and the lemma state and the GUI are updated accordingly as soon as the proof is completed (successfully or unsuccessfully). This asynchronous interaction is further justified by the fact that a lemma never changes once having come into existence¹. Incidentally, the asynchronous interaction model is also the *raison d'être* for the `pending` Simplify state.

5.1.1. Implementation

Worker Threads and Simplify Processes

Internally, whenever a lemma is sent to Simplify, a new worker thread is spun off to handle the interaction with Simplify and to update the lemma state and the GUI after the proof terminates.

Since Simplify has to be initialized with the background predicates of the formal data and state model, it is more efficient to have one continuously running Simplify (sub)process rather than to start a new process for each lemma to be proven. The continuously running Simplify process needs to be created and initialized only once, whereupon it is ready to accept incoming lemmas. Thus, in addition to the initialization cost, the overhead of creating and starting a new process is also saved for each lemma to be verified. The only slight downside of this design decision is the fact that a reinitialization of the Simplify process (e.g., in case some related user preferences are changed, cf. Section 5.2) requires terminating and restarting it.

While spinning off worker threads avoids blocking the user on the GUI level, the interaction with the Simplify process within the worker threads obviously happens in a synchronous way. I.e., when the Simplify process is busy proving a lemma, the other worker threads have to wait.

Simplify Process Pool

In order to balance the workload and reduce the turnaround time, the implementation actually maintains not only one Simplify process, but a pool of N Simplify processes (currently, $N = 3$). As illustration, imagine a complex and a simple lemma being sent (in this order) to Simplify, where the complex lemma takes a much longer time to be

¹The string representation of a lemma may change, but the actual formula to be verified does not.

proven than the simple one. With only one Simplify process the user would have to wait for the long, complex proof to be completed until the simple lemma's proof will be available. However, should there be another Simplify process that happens to be idle, the simple lemma can be handled by that process.

The Simplify processes in the pool are assigned to the worker threads in a simple round robin pattern. Each Simplify process is lazily created and initialized, at the latest possible time. In particular, a process is only created if it actually has been assigned to a lemma to be proven. This results in no loss of resources for Jive users not using Simplify.

Whenever a lemma is to be proven by a particular Simplify process, a sanity check is performed whether the process in question is still alive. This is done by sending the trivial proposition **True**. Should this fail to be proven, the process is recreated and reinitialized. Furthermore, a crash of a Simplify process while proving a lemma does not amount to Jive possibly waiting forever for a result, since the process' output stream is also terminated². In such a case, the lemma state is restored to the state prior to the Simplify proof, but the lemma is not automatically resent to Simplify.

The handling of the Simplify process pool (including creating, initializing and checking processes) is implemented in the class `jive.TPC.simplify.SimplifySessionManager`, whereas the class `jive.TPC.simplify.SimplifySession` represents a single Simplify process. The latter class is partially based on the Daikon invariant detector [12] source code.

Figure 5.1 schematically illustrates the entire setting.

Removing and Saving Proof Tree Nodes

Care has to be taken in the implementation in case a node is removed from the proof tree by a cut operation (“Cut” in the “Control” menu) while its lemma is still being proven by Simplify. In such a scenario, the result of the Simplify proof is simply discarded. The pending Simplify proof is not terminated abruptly, however, since this would involve killing the Simplify process.

Furthermore, if a user happens to save the proof tree (“Save Prooftrees” in the “File” menu) while some lemmas are still being proven by Simplify, the Simplify states of the pending lemmas are saved as the respective states prior to the Simplify proof instead of **pending**. This is to guarantee a consistent state when the proof tree is loaded back into Jive.

²At least, this is the behavior on Linux.

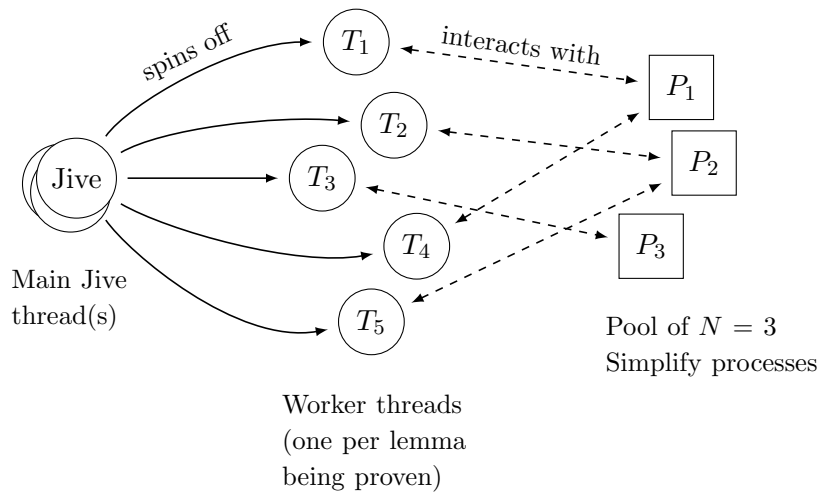


Figure 5.1.: Interaction with Simplify

5.2. User Preferences

The following is a summary of the Jive user preferences related to Simplify (cf. Figure 5.2). The default values are given in parentheses. Similar descriptions are also displayed to the user in tooltips in the preferences dialog.

Close proof obligations if proven by Simplify (false). If true, a lemma that has been proven by Simplify is treated equivalently as though proven by Isabelle. I.e., not only the Simplify lemma state, but also the lemma state w.r.t. Isabelle is set to “proven” and the proof obligation is closed. Changes to this setting affect only newly proven lemmas.

This preference reflects the user’s trust in the Simplify integration. So, a user who does not fully trust the Simplify integration can still prove all lemmas in Isabelle, even if a lemma may already have been proven by Simplify. In any case, the Simplify state (unless `initial` or `pending`) of each lemma is included as a short comment note in exported Isabelle verification condition theories and in generated Latex proofs.

Automatically send lemmas to Simplify (false). If true, a lemma is sent to Simplify as soon as it is generated (see also Section 5.3). Changes to this setting affect only newly generated lemmas.

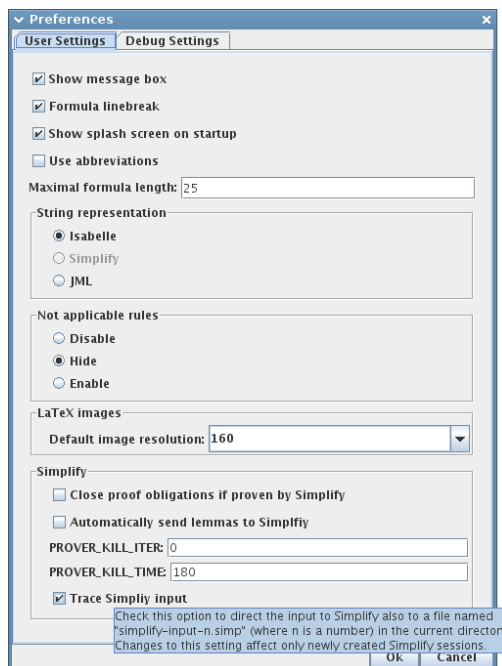


Figure 5.2.: Preferences dialog showing Simplify related options

PROVER_KILL_ITER (0). Represents the largest number of iterations for which Simplify should be allowed to run on any single lemma before giving up. A value of 0 means not to bound the number of iterations at all. Changes to this setting affect only newly created Simplify processes³.

PROVER_KILL_TIME (180). Represents the longest time period (in seconds) Simplify should be allowed to run on any single lemma before giving up. A value of 0 means to not bound Simplify at all by time. Beware that using this option might make Simplify’s output depend on the speed of the machine it is run on. Changes to this setting affect only newly created Simplify processes³.

Trace Simplify input (false). If true, the input to Simplify also directed to a file named “simplify-input-*n*.simp” (where *n* is a number) in the current directory. This is intended primarily for debugging when Simplify fails. Changes to this setting affect only newly created Simplify processes.

³This is due to the fact that the values for the PROVER_KILL_ITER and PROVER_KILL_TIME parameters have to be passed as environment variables to a Simplify process at creation time. In consequence, changing the values has no effect on already created Simplify processes. In most cases, this means that Jive has to be restarted for the changes to take effect (with the exception of not yet created Simplify processes).

The implementation is prepared for an easy addition of other Simplify environment variables besides `PROVER_KILL_ITER` and `PROVER_KILL_TIME`, perhaps new ones introduced in future versions of Simplify.

5.3. Triggering a Simplify Proof

The command “Send all Lemmas to Simplify” in the “File” menu (Figure 5.3) can be used to send *all* currently available lemmas with Simplify state `initial` to Simplify. As ancillary information, the number of such lemmas is also shown in the menu item. If there are no lemmas with state `initial`, the menu item is disabled accordingly.

In addition, there is a popup menu with items “Send Lemma to Simplify” and “View Lemma in Simplify Syntax” associated with every proof tree node (cf. Figure 5.4). Both items are enabled only if the proof tree node actually contains a lemma. In contrast to the batch command “Send all Lemmas to Simplify”, the item “Send Lemma to Simplify” in the popup menu allows the user to prove lemmas selectively in a one-by-one fashion. Clicking on the item “View Lemma in Simplify Syntax” brings up a modal dialog box containing the lemma as unparsed to Simplify syntax, which may be copied to the clipboard for a further analysis or debugging.

As a third option, the lemmas can automatically be sent to Simplify as soon as they are generated. This feature is controlled by the “Automatically send lemmas to Simplify” user preference (cf. Section 5.2).

If a lemma is either proven or disproven by Simplify, there is no need to resend it to Simplify at a later time. So, the command “Send Lemma to Simplify” in the popup menu is disabled for a lemma with state `proven` or `disproven`. In contrast, there are cases where it makes sense to resend a `timed_out` lemma to Simplify. E.g., the user might want to increase the corresponding Simplify user preference parameters and give the proof another try⁴. Consequently, the “Send Lemma to Simplify” command in the popup menu changes to “Resend Lemma to Simplify” and stays enabled for a lemma with state `timed_out`. However, in the current implementation, the batch command in the “File” menu still handles only lemmas with state `initial`. One might consider introducing an additional batch command for lemmas with state `timed_out`, or having only one batch command for both states. This is only a minor design issue.

As a last remark, whenever a lemma is sent to Simplify and whenever a Simplify proof finishes, a corresponding informative message is displayed on the message panel in the bottom part of the Jive main window (cf. Figure 5.4).

⁴As noted in Section 5.2, this may entail relaunching Jive.

5.3. Triggering a Simplify Proof

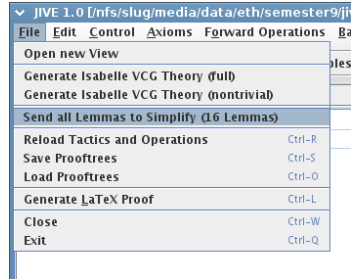


Figure 5.3.: “File” menu

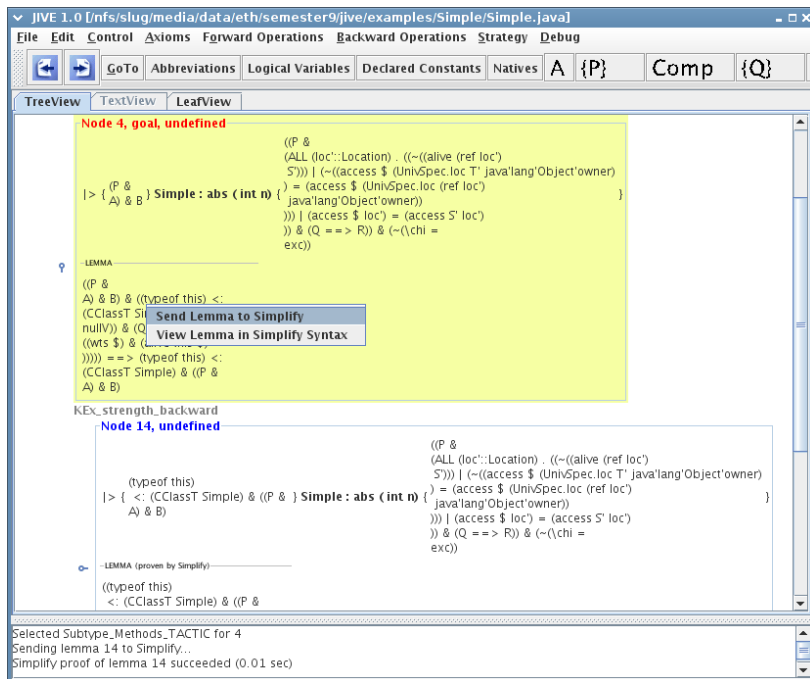


Figure 5.4.: Jive’s main window

5.4. Future Work

The current implementation completely hides the running Simplify processes from the user. In future versions, one might think of giving the user explicit control over the currently running Simplify processes through a graphical monitor/control panel that lists each Simplify process, its current state (e.g., “proving lemma n ” or “idle”) and perhaps some initialization data (e.g., “PROVER_KILL_TIME=300”). The user could then also be allowed to restart and reinitialize Simplify processes without having to restart Jive.

For another usability enhancement, imagine that a large number of lemmas have been sent to Simplify (using the batch command in the “File” menu), but some lemmas failed to be proven, as indicated on the Jive message panel. Instead of the user having to manually scroll through the proof tree in order to find a particular failed lemma, a mechanism to directly jump to a given proof tree node, perhaps triggered by a click on the corresponding failure message in the message panel, would come in handy.

Finally, as mentioned before, the incorporation of Simplify into the text view is still due.

Bibliography

- [1] Meyer, J. and Poetzsch-Heffter, A., *An architecture for interactive program provers*, in S. Graf and M. Schwartzbach, editors, Tools and Algorithms for the Construction and Analysis of Systems, number 1785 in Lect. Notes Comp. Sci., pages 63-77, Springer, Berlin, 2000.
- [2] Meyer, J. and Müller, P. and Poetzsch-Heffter, A., *The Jive system – implementation description*, 2000.
- [3] Nipkow, T. and Paulson, L. C. and Wenzel, M., *Isabelle/HOL: A proof assistant for higher-order logic*, Springer, May 19, 2006. <http://isabelle.in.tum.de/>
- [4] Detlefs, D. L. and Nelson, G. and Saxe, J. B., *Simplify: A theorem prover for program checking*, Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.
- [5] Leavens, G. and Cheon, Y., *Design by contract with JML*, Java Modeling Language Project, 2003. <http://www.jmlspecs.org/>
- [6] Detlefs, D. L. and Leino, K. R. M. and Nelson, G. and Saxe, J. B., *Extended static checking*, Technical Report 159, Compaq SRC, 1998.
- [7] Kiniry, J. R., et al., *The logics and calculi of ESC/Java2*, November 2004. <http://secure.ucd.ie/products/opensource/ESCJava2/>
- [8] Poetzsch-Heffter, A. and Müller, P., *Logical foundations for typed object-oriented languages*, in D. Gries and W. De Roever, editors, Programming Concepts and Methods (PRO-COMET), 1998.
- [9] Poetzsch-Heffter, A. and Gaillourdet, J.-M. and Rauch, N., *Soundness and relative completeness of a programming logic for a sequential Java subset*, Internal Report.
- [10] Schäfer, J., *Generating order-sorted data types in Java*, Internal Project Report, 2004.
- [11] Barnett M. and Leino, K. R. M. and Schulte, W., *The Spec# programming system: An overview*, in CASSIS, Lecture Notes in Computer Science, Springer, 2004. <http://research.microsoft.com/specsharp/>
- [12] *The Daikon invariant detector*. <http://pag.csail.mit.edu/daikon/>

All WWW links as of September 10, 2006.

A. Appendix

A.1. Terms and Formulas in Simplify

The following describes Simplify's term and formula (or predicate) model in the Katja specification language.

A.1.1. Terms

```
Term = Constant
      | Variable
      | FunctionApplication

TermList * Term

Constant = TTrue() // @true
          | Integer
          | String

Variable = String

VariableList * Variable

FunctionApplication = FuncAppl ( Function func, TermList args )

Function = TPlus() // math operator +
          | TMinus() // math operator -
          | TTimes() // math operator *
          | String // uninterpreted function
```

A.1.2. Formulas

```
Formula = True()
         | False()
         | Relation ( Term left, RelOp op, Term right )
         | FNot ( Formula formula ) // negation
         | BinaryFormula ( Formula left, FormulaOp op, Formula right )
         | BindingFormula ( BindingOp op,
                           VariableList variableList,
                           Formula formula )

RelOp = REq()          // equality ==
       | RNeq()       // inequality !=
       | RLess()      // comparison <
       | RLessEq()    // comparison <=
       | RGreater()   // comparison >
       | RGreaterEq() // comparison >=

FormulaOp = FAnd()     // logical and /\
          | FOr()      // logical or \/
          | FImplies() // logical implication ==>
          | FIff()     // logical equivalence <==>

BindingOp = Forall() // universal quantifier
          | Exists() // existential quantifier
```

A.2. Terms and Formulas in Jive

A.2.1. Terms

The following is an excerpt of the Katja specification file `term.katja` as of September 10, 2006.

```
Term = Constant
      | Variable
      | FunctionApplication
      | UnaryOpTerm ( UnaryOp op, Term t )
      | BinaryOpTerm ( Term term1, BinaryOp op, Term term2 )
      | ConditionalTerm ( Term condition, Term thenExpr, Term elseExpr )
      | SetTerm ( TermList elems )

TermList * Term

Constant = ByteLiteral ( Byte value )
          | ShortLiteral ( Short value )
          | IntegerLiteral ( Integer value )
          | NatLiteral ( Integer value )
          | DeclaredConst ( String theory, String name )
          | True ()
          | False ()

Variable = ProgVar ( String name )
          | LogicalVar ( String name )
          | Dollar ()
          | Chi ()

FunctionApplication = FuncAppl ( DeclaredConst func, TermList args )
                   | Tuple ( TermList entries )

UnaryOp = TNot() // Java boolean negation !
        | TBitCompl() // bit complement
        | TUPlus() // unary plus +
        | TUMinus() // unary minus -

BinaryOp = TLogicalAnd() // strict AND &
         | TLogicalOr() // strict OR |
```

```
| TConditionalAnd() // lazy AND &&
| TConditionalOr() // lazy OR ||
| TEq()             // equality ==
| TLess()           // comparison <
| TLessEq()         // comparison <=
| TLeftShift()     // left shift <<
| TRightShift()    // right shift >>
| TUnsignedRightShift() // unsigned right shift >>>
| TPlus()           // math operator +
| TMinus()          // math operator -
| TTimes()          // math operator *
| TDiv()            // math operator /
| TMod()            // math operator %
| TBitAnd()         // bitwise AND &
| TBitOr()          // bitwise OR |
| TBitXOr()         // bitwise XOR ^
| TElement()        // JML subset relation \in
| TInstanceOf()     // JML instanceof operator
| TSubtype()        // JML: <:
| TProperSubtype() // JML: <<:
```

A.2.2. Formulas

The following is an excerpt of the Katja specification file `formula.katja` as of September 10, 2006.

```
Formula = IsTrue( Term t )           // transition Term -> Formula
        | FNot ( Formula formula ) // negation
        | BinaryFormula ( Formula left, FormulaOp op, Formula right )
        | BindingFormula ( BindingOp op,
                          BindingList bindingList,
                          Formula formula )

FormulaOp = FAnd() // logical and /\
          | FOr() // logical or \/
          | FImplies() // logical implication ==>
          | FIff() // logical equivalence <==>
          | FNotIff() // logical inequivalence <!=>

BindingOp = Forall() // universal quantifier
          | Exists() // existential quantifier

BindingList * Binding

Binding (LogicalVar lv, Type t)
```
